Code Variants and Their Retrieval Using Knowledge Discovery Based Approaches

by
Venkatesh Vinayakarao

Under the Supervision of
Dr. Rahul Purandare
Indraprastha Institute of Information Technology, Delhi

Indraprastha Institute of Information Technology, Delhi
April, 2018

Code Variants and Their Retrieval Using Knowledge Discovery Based Approaches

by
Venkatesh Vinayakarao

Submitted
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Indraprastha Institute of Information Technology, Delhi
April, 2018

Certificate

This is to certify that the thesis titled - **"Code Variants and Their Retrieval Using Knowledge Discovery Based Approaches"** being submitted by **Venkatesh Vinayakarao** to Indraprastha Institute of Information Technology, Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by him under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

[illegible]

Dr. Rahul Purandare
Department of Computer Science
Indraprastha Institute of Information Technology, Delhi
April, 2018

Acknowledgments

I would like to thank my research advisor Dr. Rahul Purandare for the continuous support during my PhD study at IIIT-Delhi. Rahul's support, suggestions and advice contributed significantly to this work. The provided scientific freedom and guidance were instrumental to the success of this thesis.

I would also like to express my gratitude to my industry mentor Dr. Aditya Nori of Microsoft Research who provided valuable guidance, support and feedback. Time spent at Microsoft Research, Cambridge with Aditya was invaluable for my progress as a mature researcher.

I would like to thank Dr. Anita Sarma for her valuable inputs and continuous support throughout my PhD work.

Discussions with the members of my research committee, Dr. Ponnurangam Kumaraguru and Dr. Pushpendra Singh helped me a long way in shaping up my PhD. Beyond just reviewing my PhD progress, they were present whenever I needed any advice irrespective of whether they were professional or personal.

I would like to acknowledge my co-authors of papers published during my PhD. It was a great experience working with all of you: Shuktika Jain, Saumya Jain, Devika Sondhi, Vishal Raj Dutta, Prashant, Sai Prathik, Ridhi Jain, Sumit Keswani and Aditi Mittal. Thank you.

Thanks to all my previous and current lab-mates at IIIT-Delhi. Your cheerful presence in the lab made it a very inviting place. Special thanks to my academic sisters, Dhriti Khanna, Ridhi Jain and Devika Sondhi.

Many thanks to my friends in Delhi who supported me in my pursuit for academic excellence. Without you guys, I would not have survived the five years of my research. Deeply inter-twined with the PhD memories are the over the breakfast chat, gossips over lunch, a leisurely tea and the after-dinner walks. Haroon Rashid, Rekha Tokas and Sangeet Kochanthara deserve a special mention.

I would also like to thank Dr. Srikanta Bedathur for without him, I would not have been in IIIT-Delhi.

Thanks to admin and support staff at IIIT-Delhi. They made my life so easy here that I could focus only on my research.

I am a recipient of Prime Ministers Fellowship Scheme for Doctoral Research, a public-private partnership between Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India and Confederation of Indian Industry (CII). My gratitude to CII and SERB. Particularly, thanks to Dr. Kohli, Neha and Shalini for supporting me with all the fellowship related queries. Thanks to Microsoft Research for funding a part of my fellowship.

Special thanks to my mother, father and sisters for their encouragement and love throughout these years. Throughout the tough times, you have been with me for which no words of gratitude will be sufficient. Your dreams of seeing a "Dr." prefix in front of my name will soon come true.

To my dear wife Bhargavi and lovely son Tarun, not a day goes without thinking about you. There is a special place in my heart for you. I dedicate this thesis to you.

List of Publications

1. Ridhi Jain, Sai Prathik Saba Bama, Venkatesh Vinayakarao and Rahul Purandare. A Search System for Mathematical Expressions on Software Binaries. In the Proceedings of The 15th International Conference on Mining Software Repositories (MSR 2018), Sweden. [Chapter 6]

2. Venkatesh Vinayakarao, Anita Sarma, Rahul Purandare, Shuktika Jain and Saumya Jain. ANNE: Improving Source Code Search using Entity Retrieval Approach. In the Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM 2017), UK. [Chapter 5]

3. Venkatesh Vinayakarao, Rahul Purandare and Aditya Nori. Structurally Heterogeneous Source Code Examples from Unstructured Knowledge Sources. In the Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2015), India. [Chapters 3 and 4]

4. Venkatesh Vinayakarao. Spotting familiar code snippet structures for program comprehension. In the Proceedings of the 2015 10th Meeting on Foundations of Software Engineering, (ESEC/FSE 2015), Italy. [Chapters 3 and 4]

Abstract

Code variants represent alternative implementations of a code snippet, where each alternative provides the same functionality, but has different properties that make some of them better suited to the overall project requirements. Developers routinely need to analyze existing code, find better reuse alternatives, and look to develop high-quality code that meets some desired properties. However, searching for such code variants over the web has several challenges. In this dissertation, we address this problem.

This dissertation presents new techniques to search for code variants. Classical program analysis techniques do not scale well to analyze partial programs at web-scale. Hence, we apply search techniques to mine code variants using human annotated natural language descriptions found in the posts of Stack Overflow[1] (SO) which is a popular discussion forum. Here, we make four major contributions.

Unlike clones and examples, existing literature lacks a rigorous characterization of code variants. So, as our first contribution, we present a characterization of code variants where we discuss the code context, desired properties, and types of variants along with implications for tool builders.

With this knowledge about variants, we propose techniques to search for variants in SO, as our second major contribution. We propose a novel structural model for source code which is based on developers' perspective of similarity. To leverage the text and code components that we index from SO, we adapt an existing state of the art term-weighting method to propose a Multi-Component Multi-Aspect Term Frequency - Inverse Document Frequency (MCMATF-IDF) model to retrieve code variants.

Existing text-retrieval models do not work well on source code. Expressing natural language queries on source code is an open problem. Many query terms in natural language have multiple surface forms in source code. We address this problem by perceiving source code as a collection of entities. This becomes our third major contribution.

Further, as a bottleneck to the success of our approaches, we notice that our work depends on parsing code snippets in SO. We observe that only 31.3% of code snippets in SO parse. Hence, in our fourth contribution, we apply grounded theory approach to study these parsing problems. Based on this study, we develop a tool which increases the code snippets that can generate Abstract Syntax Trees for 63% of the code snippets in SO.

Overall, the ability to perform semantic search over source code snippets assisted by developer knowledge in the form of discussion forum data opens up a new way to solve several important problems. It can lead to improvements in a variety of software engineering tasks and tools such as semantic clone detection, code comprehension and defect detection. Apart from supporting software engineering applications, as future work, we plan to explore enhancing static analysis over source code snippets using the data from discussion forums.

[1]http://stackoverflow.com/

Contents

1 Introduction **1**
- 1.1 Thesis Statement . . . 3
- 1.2 Contributions . . . 3
- 1.3 Outline of Dissertation . . . 4

2 Background **5**
- 2.1 Code Search . . . 5
  - 2.1.1 Indexing Source Code . . . 7
  - 2.1.2 Retrieval Models for Source Code . . . 9

3 Code Variants **13**
- 3.1 Introduction . . . 13
- 3.2 Background and Related Work . . . 15
  - 3.2.1 Background on the Types of code snippets . . . 15
  - 3.2.2 Redundancies in source code . . . 16
  - 3.2.3 Variants: A missing link . . . 17
- 3.3 Research Methodology . . . 17
- 3.4 RQ1: What are Code Variants? Are they different from the known code snippet types? . . . 19
- 3.5 RQ2: What are the aspects of variants that developers consider while describing variants? . . . 20
  - 3.5.1 Code Context . . . 20
  - 3.5.2 Desired Properties . . . 23
- 3.6 RQ3: Do Developers Seek Variants? . . . 27
  - 3.6.1 Nature of Variants . . . 27
  - 3.6.2 Are variants language dependent? . . . 31
- 3.7 Implications of Variant Characterization . . . 32
  - 3.7.1 Tool Builders . . . 32
  - 3.7.2 Researchers . . . 32
- 3.8 Threats to Validity . . . 33
- 3.9 Summary . . . 33

4 Towards Searching for Code Variants **35**
- 4.1 Introduction . . . 35
- 4.2 jSense . . . 38

4.2.1 jSense Retrieval Model - MC-MATF . . . 38
4.2.2 Structural Model . . . 39
4.2.3 Refining the Structural Model . . . 40
4.3 Implementation . . . 44
4.3.1 Building the Repository . . . 45
4.3.2 Querying the Repository . . . 46
4.4 Evaluation . . . 46
4.4.1 Building the Repository . . . 47
4.4.2 Querying the Repository . . . 49
4.5 Threats to Validity . . . 52
4.6 Related Work . . . 52
4.7 Summary . . . 54

5 Improving Code Search using Entity Retrieval 55
5.1 Introduction . . . 55
5.2 Motivation . . . 56
5.2.1 Formative Study . . . 56
5.2.2 Applications . . . 58
5.2.3 Problem Overview . . . 58
5.3 Definitions . . . 59
5.4 Approach . . . 60
5.4.1 Entity Discovery . . . 60
5.4.2 Entity Profile Construction . . . 61
5.4.3 Entity Linking . . . 62
5.5 Evaluation . . . 64
5.5.1 Entity Discovery . . . 64
5.5.2 Entity Profile Construction . . . 66
5.6 User Study . . . 67
5.6.1 Study Design . . . 67
5.6.2 Results . . . 69
5.7 Limitations and Threats . . . 71
5.8 Summary . . . 71

6 Reducing the Parsing Problems in Stack Overflow 73
6.1 Introduction . . . 73
6.2 Background and Terminology . . . 75
6.3 Research Method . . . 76
6.4 Data Analysis . . . 77
6.4.1 Issues due to Developer Behavior . . . 77
6.4.2 Platform Related Issues . . . 80
6.4.3 Language Related Issues . . . 81
6.5 Results . . . 81
6.6 Building a Classifier to Validate Results . . . 82
6.7 Implementation and Evaluation of Classifier . . . 85
6.7.1 Parameter Tuning . . . 86

6.7.2 Comparative Evaluation . . . 87
6.8 Validating the Study Results using the Classifier . . . 88
6.8.1 Validation . . . 88
6.8.2 Effectiveness of jMechanic . . . 89
6.9 Related Work . . . 89
6.10 Threats to Validity . . . 90
6.11 Summary . . . 90

7 Conclusion and Future Work 92
7.1 Resources . . . 93
7.2 Future Work . . . 93
7.2.1 Applications of Code Variants . . . 93
7.2.2 Modeling Tasks as Search Problems over Source Code . . . 94
7.2.3 Searching in Software Binaries . . . 95

A User Study - Searching for Variants 97

List of Figures

1.1 Summary of contributions of this dissertation. (1) First, we characterize a class of code snippets called Code Variants. (2) We build jSense tool to search for code variants. (3) We improve the search with an entity based approach in a tool named ANNE. (4) Finally, we investigate the parsing and compilation issues in partial programs and build a tool jMechanic to solve some of the identified problems. 2

2.1 Growing information needs lead to technology improvement. 6
2.2 A typical IR system which takes queries and returns ranked results. Unstructured content is indexed for faster and efficient retrieval. 7
2.3 Why MATF? SO posts carry developer discussions (answers) that have 14 to 66 distinct terms and 11 to 118 terms overall in them. The distinct terms count range from 0 to 2053. The total terms range from 0 to 8089. This figure considers only those posts where the counts of discussions are above 10,000. Due to the significance that posts carry due to the presence of distinct terms, we use MATF as our retrieval model. 11

3.1 Reuse in source code happens at different levels. This work focuses on code variants. 14
3.2 Differences between clones, simple variants and complex variants in terms of desired properties. 29
3.3 Distribution of variants across context types and desired properties over multiple languages. Diction variants are prominent in projects of all programming languages. 31

4.1 Overview of jSense design. This is a two step process: 1) We build a repository of structurally heterogeneous implementations, and 2) Use this repository for suggesting implementation choices for the identified topic. 37
4.2 A snapshot of the tool we used in our study of structural similarity is shown here. Users categorized the given snippet into 1 to 10 types based on their judgment. 41
4.3 We build a repository of structurally heterogeneous implementations for a given set of topics. We use jSense retrieval model to find relevant posts from SO. We use jSense structural model for computing code similarity. 45

4.4 While de-duplicating the results, for different values of ψ, we note the corresponding F_1Score for each topic. We observe that a cut-off of 0.4 yields best results. 48

5.1 Anne annotates input code snippets, line by line with natural language terms. These annotations help keyword based search engines to address NL queries. 58
5.2 Entity discovery subsystem works on NL text using parts-of-speech (PoS) approach. We use seed entities to discover more entities that fit into the same grammatical sequence. 60
5.3 Entity linker subsystem works line by line on the input code, to find matching entity profiles. Entity names whose profiles match are stamped across the line, as shown in the example. 63
5.4 Tagged versions of the tasks (A) Enum and (B) IncDec that were provided to the participants in the user study. 64
5.5 The goldset for evaluation is created from SO posts by mixing posts that contain seed entities in the title with those that do not have them. 66
5.6 Code search tool used for giving feedback to student assignments. This tool allows us to toggle tagging on and off for evaluation. 69

6.1 Overview of our study of variants. (1) We extract code snippets from SO. (2) We attempt to parse those snippets using eclipse JDT. (3) For snippets that do not parse, we analyze using a GT approach. (4) We propose a list of issues that dominate while parsing. (5) We build a classifier using a training set from our study. (6) We apply this classifier on entire SO data. (6) Thus, we validate our theory on entire SO. 74
6.2 Discovered concepts and categories. Concepts in the Developer category are further classified into the subcategories of Supplements, Suppresions and Substitutions. Some of the concepts are introduced intentionally by developers for the purposes of brevity and focus. These are marked as "intentional" in this figure. Our focus is on extraction related issues instead of curation issues. We have distinguished them here. We build a classifier for three of these concepts namely Outputs, HTML Elements and Ellipses. We show that this helps jMechanic tool to improves on its ability to parse more snippets. 76
6.3 (a) Concepts and categories discovered using grounded theory approach. (b) When validated against a random sample of 500 snippets, we observe strong correlation. 80
6.4 jMechanic parse issue detection uses a classifier which learns from Mixed MATF weights. 83
6.5 A component weight setting of 0.2 works best for our work. 86
6.6 Examples of Threshold Parameter Estimation 86
6.7 We compare the results of manual study on limited snippets with the results of the classifier on entire SO. We find that the correlation is significant for all the classes. Comparison with Naive Bayes shows that our classifier performs 45% better. 87

List of Tables

2.1 Precision@10 of existing code search engines. 6

3.1 Code variants are discussed in defect descriptions of Apache Math, a popular open source project. 14
3.2 Fundamental differences exist between semantic clones, code examples and code variants. 16
3.3 Projects used to characterize variants. 18
3.4 Code contexts are primarily described using one or more of these four types. We map defect descriptions containing variants to these types. 22
3.5 Examples of developer discussions taken from Eclipse project describing the desired properties in code variants. 23
3.6 Discussions (with referenced defect from the issue tracker) from the dataset capturing the following: a) the desired properties, and b) the code contexts across variants. (S# captures the key statements from the discussion) 25
3.6 Discussions (with referenced defect from the issue tracker) from the dataset capturing the following: a) the desired properties, and b) the code contexts across variants. (S# captures the key statements from the discussion) 26
3.6 Discussions (with referenced defect from the issue tracker) from the dataset capturing the following: a) the desired properties, and b) the code contexts across variants. (S# captures the key statements from the discussion) 27
3.7 Examples of developer discussions describing the variant types.(S# captures key statements from the discussion.) 28
3.8 Volume of variant discussions in open source projects depends on the project domain. T_v, Alg, RO, D_p, D_m are the counts of: total, algorithmic, RO and pure-diction and mechanics variants respectively. 28
3.9 Summary of discussion counts per variant type. Total refers to the total number of discussions. $\%_v$ is the Total as percentage out of 1500 discussions. $\%_d$ is the percentage out of 426 variants. 30

4.1 Some examples of factorial implementations. 36
4.2 We have used four reduction rules. Here, we show two code snippets that reduce to an identical snippet after the application of reduction rule. 43
4.3 Some examples of jSense structural representation after the application of reductions and structure flattening. 44
4.4 A comparison of snippets retained by our approach against a developer's hand-picked set of implementations, for the *"factorial"* topic. 49

4.5 We evaluate jSense by comparing the results with a goldset. GS represents the gold set. JS_b and CE_b represent the baseline versions of jSense tool and CodeExchange CE respectively. 50
4.6 We extract implementations with an average precision of 0.92 compared to the 0.69 of CodeExchange (CE). jSense (JS) improves recall from 0.29 CodeExchange (CE)to 0.71 (JS). 51
4.7 Ability to differentiate the components allows MC-MATF model to outperform the other retrieval models. 51

5.1 P@10 of existing code search engines for NL queries containing programming concepts. 56
5.2 Formative study on 25 real industry developers indicates that this research will be useful. 57
5.3 Patterns and frequencies for ***conditional*** in Java snippets found in SO. 61
5.4 No. of entities discovered is related to the length of PoS patterns considered in our approach. Longer patterns produce fewer entities that exhibit higher level of similarity to seed entity. 65
5.5 Performance of ANNE Entity Discovery module. Experiments were carried out on a gold set with 1:1 noise and 1:3 noise. F_1 indicates the F_1-score and —E— stands for the number of entities discovered. 65
5.6 Manually computed precision@4 and the top pattern discovered for some of the entities. We use top four patterns while annotating source code. 67
5.7 Two factor design that counterbalances the treatment and the task. 67
5.8 Descriptive statistics of number of incorrect assignments found by participants. 68
5.9 Terms used to calculate correctness and completeness scores for a submission S. 68
5.10 Correctness/Completeness metrics of participants with std. deviation in parentheses. 70
5.11 Time taken to complete (in minutes) assessment for tagged and untagged versions. 70

6.1 Precision, Recall and F_1 Score of the classifier. Our classifier gives good results for all cases where term saliency can be observed. 86
6.2 Threshold Values and Descriptive Terms extracted using the *Extended MATF Model for Source Code*. 87
6.3 Statistics on the parseable and erroneous snippets in Java tagged posts in Stack Overflow. 88
6.4 The percentages represent the fraction of erroneous snippets that suffer from the given issue. GT Study represents numbers from our initial study. GT Validation shows the results from manual validation. Last column gives the results observed after running the classifier on entire dump. 88

Glossary

ANNE	:	ANNotation Engine
API	:	Application Programming Interface
ASP	:	Active Server Pages
AST	:	Abstract Syntax Tree
ATF	:	Augmented Term Frequency
BM25	:	Best Match 25
BRCG	:	Branch-Reserving Call Graph
CFG	:	Control Flow Graph
CII	:	Confederation of Indian Industries
CRF	:	Conditional Random Field
DARPA	:	Defense Advanced Research Projects Agency
GraPacc	:	GRaph-based, Pattern oriented, Context-sensitive Code Completion
GT	:	Grounded Theory
IDE	:	Integrated Development Environment
IDF	:	Inverse Document Frequency
FCA	:	Formal Concept Analysis
IR	:	Information Retrieval
JDT	:	Java Development Tools
KB	:	Knowledge Base
LRTF	:	Length Regulated Term Frequency
LDA	:	Latent Dirichlet Allocatio
LSA	:	Latent Semantic Analysis
LSI	:	Latent Semantic Index
MATF	:	Multi Aspect Term Frequency
MC-MATF	:	Multi Component Multi Aspect Term Frequency
MOSS	:	Measure Of Software Similarity
MUSE	:	Mining and Understanding Software Enclaves
NER	:	Named Entity Recognizer
NL	:	Natural Language
PA	:	Program Analysis
PDG	:	Program Dependence Graph
PoS	:	Parts of Speech
RITF	:	Relative Intra-Document Term Frequency
RO	:	Resource-Oriented
RQ	:	Research Question
SE	:	Software Engineering
SERB	:	Science and Engineering Research Board
SITIR	:	SIngle Trace and Information Retrieval
SO	:	Stack Overflow
SSI	:	Structural Semantic Indexing
SVD	:	Singular Value Decomposition
TA	:	Teaching Assistants

TF	:	Term Frequency
VSM	:	Vector Space Model

Chapter 1

Introduction

Developers seldom write code from scratch [1, 2]. While considering code snippets for reuse, they often have to reason among the benefits and drawbacks before making a selection. Our work is motivated by the lack of tool support for this code reuse task. We posit that, developers seek an alternate implementation choice that fits into the current code context and possesses the desired properties. We call a code snippet that forms such an alternate implementation choice, as a code variant. In this work, we characterize, and build tool support for searching code variants.

We show that code variants are fundamentally different from other code snippet types such as clones, simions, idioms and examples. Clones were originally defined as redundant snippets introduced due to a copy and paste activity [3]. While syntactic clones which are also referred to as Type I, II and III clones usually have no specific desired property in them to replace existing code snippets, code variants are closer to semantic clones. Unfortunately, semantic clones do not have a consistent definition. It is used in different contexts to address a variety of snippet types such as wide-miss clones [4], interleaved clones [5], and high-level concept clones [4]. Yet, in all these cases, semantic clones are also considered redundant copies of code which are preferred to be refactored out of the current system. In contrast, variants are sought by developers to replace existing snippets. On a similar note, we also find code examples, simions and idioms to be different when compared to variants.

Although the term "variants" is used loosely by developers and researchers, literature lacks a clear definition of variants. We use this opportunity to characterize code variants. Code variants are closely tied to the current code context and desired properties. We investigate the code context and desired properties in more detail. We envision to support the developers by automating the task of searching for variants. As a first step in the direction of searching for variants, we show that a search for implementation choices can be performed using the data from a popular discussion forum, Stack Overflow (SO).

In spite of search being the most common activity among software engineers [6], searching for source code, unlike text retrieval, is a less explored problem. Sim et al. [7] show that a general purpose search engine such as Google performs better for code snippet search compared to specialized code search engines. Source code presents unique challenges in different stages of the retrieval process such as query understanding, indexing and adapting the retrieval models.

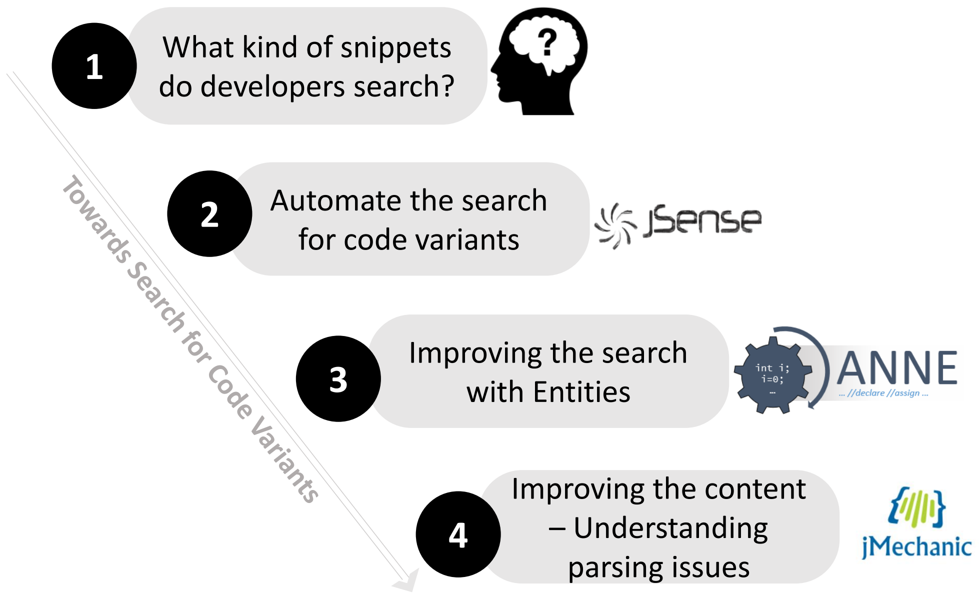

1
What kind of snippets do developers search?
2
Automate the search for code variants
jSense
3
Improving the search with Entities
ANNE
4
Improving the content – Understanding parsing issues
jMechanic
Towards Search for Code Variants

Figure 1.1: Summary of contributions of this dissertation. (1) First, we characterize a class of code snippets called Code Variants. (2) We build jSense tool to search for code variants. (3) We improve the search with an entity based approach in a tool named ANNE. (4) Finally, we investigate the parsing and compilation issues in partial programs and build a tool jMechanic to solve some of the identified problems.

Expressing queries for search even in Natural Language (NL) suffers from the problem of mismatch in the NL vocabulary and source code representation. For instance, the term "array" in NL query does not match its representation of "`[ ]`" in Java. In this work, we present a technique to discover such mappings between syntactic forms and natural language terms representing programming concepts. We use the questions and answers in SO to create this mapping. We implement our approach in a tool called Anne.

While working with code snippets in SO, we observe that only 31.3% of the Java code snippets parse into Abstract Syntax Trees (AST). We study parsing issues on Java code snippets in SO using a grounded theory approach. Specifically, we discover the dominant reasons that cause parsing problems. We build a classifier to validate our findings. We implement some of our findings in a tool named jMechanic. Our tool, jMechanic, increases the percentage of parseable Java code snippets in SO from 31.6% to 63.3%.

Through our experiments, we observe that text retrieval models do not work well for source code. There are primarily two challenges. First, we need to index the source code structure and semantics. Most existing text retrieval models index the words in the content. Similarly, code search engines will be limited to using identifiers and comments if they were to only index the tokens in source code. We propose a selective set model for indexing the structural elements in source code snippets. We do not require the snippets to compile without errors. The next problem is to arrive at a relevance and ranking score for this indexed content. We use not only source code snippets in SO, but also the text in developer discussions available surrounding the snippet in SO, to retrieve variants. Towards this end, we take a state-of-the-art term frequency based retrieval model named Multi-Aspect Term

Frequency (MATF) model and adapt it to retrieve implementation choices. We call this the Multi Component MATF model.

Our approach is inspired by the advances of two major fields of research, namely Knowledge Discovery and Mining Software Repositories. Knowledge discovery refers to the field that is concerned with mining useful knowledge from data. SO is a popular discussion forum containing 1.47 Million Java code snippets and discussions around them. We mine SO to construct a knowledge base containing code snippets, their structural representations and potential topic that they implement which is inferred from the textual discussion between users of SO. We use this knowledge base to retrieve implementation choices.

In summary, first, we identify and characterize code variants. We build a tool named jSense to search for variants. We improve the search tool by addressing the vocabulary mismatch problem using an entity based approach. We develop an Annotation Engine (ANNE) as a result of this effort. Finally, as our approach depends on our ability to analyze partial programs, we understand and resolve parsing issues and compilation issues. We build a tool jMechanic for this purpose. Figure 1.1 summarizes the contributions of this dissertation.

1.1 Thesis Statement

This dissertation confirms the following thesis:

Thesis Statement

Knowledge discovery based approaches can be used to search an important class of code snippets called code variants with satisfactory precision and recall.

More precisely,

TS-1 Code variants are an important class of code snippets particularly of interest to the software engineering community.

TS-2 Knowledge discovery driven approaches leveraging the big code available in the form of developer discussions in forums such as Stack Overflow can be used to build search engines for code variants with satisfactory precision and recall.

1.2 Contributions

The contributions of this work are:

1. Characterization of code variants based on inspection of developer discussions in SO and bug reports in 15 open source projects. Specifically, we answer the following research questions:
    - What are code variants? Are they different from known forms of code snippets such as clones, examples, idioms and simions?

    - What are the aspects of variants that affect the search for variants?
    - Do developers seek variants?

   We also discuss the implications of our observation to tool builders and researchers.

2. Based on our understanding of variants, we propose an approach to search for variants. The major contributions are the following:

    - A knowledge base driven search approach to identify the given code snippet, and show its alternative implementations.
    - We investigate and adapt the state of the art information retrieval models to build a knowledge base of code snippet structures.
    - We propose an improved structural representation of source code structure to enable us compute relevance of given snippet with the knowledge base.

3. To improve the search for source code, we propose a technique to map lines of source code with relevant programming concepts, so as to support code search engines for NL queries. This allows the users to query on programming concepts using NL terms, and need not recall the exact syntactic terms or patterns.

4. Our approach to search for variants depends on AST extracted from SO posts. We observe that only 31.3% of snippets get parsed into ASTs. To improve this aspect, we study the parsing errors in SO. Our major contributions are:

    - We answer the RQ: What are the dominant issues plaguing Java code snippets from SO that extraction tools must address to make them parseable code snippets?
    - We implement a tool named jMechanic which resolves the dominant parsing problems. Using this tool, we show that number of snippets that can be parsed from SO can be increased from 31.6% to 63.3%

1.3 Outline of Dissertation

The rest of this dissertation is organized as follows. We give the necessary background on code search in Chapter 2. Chapter 3 gives the characterization of variants. Chapter 4 explains our approach to search for variants. One of the challenges in searching for source code is the mismatch of query terms in natural language and their corresponding pattern in source code. We address this in Chapter 5. In this process, we also show a way to apply entity retrieval on source code. Our work so far, depends on our ability to parse code snippets in SO. In Chapter 6, we share our observations from a grounded theory study of parsing issues in SO. We use this knowledge to increase the number of usable snippets in SO. Finally, conclusion and future work are in Chapter 7.

Chapter 2

Background

With the growing volume of information and information needs, technology support to retrieve information has become indispensable (Figure 2.1). Ever since 300 BC., we have had information to index in the form of libraries. The first digital library project was project Gutenberg, established in the 1970. Google index grew to 130 Trillion pages in November 2016. As on April 2018, Google reports that it indexes more than 100,000,000 Gigabytes of data. This data includes text, videos, images and various other formats. We focus on Information Retrieval (IR) of a specific type of content, namely, source code. In 2017, Github reached 24 Million developers and 67 Million repositories[1].

2.1 Code Search

Developers are searching for source code [7, 8, 9]. Different information needs give rise to a variety of query types. Developers issue structural queries within the IDE for navigational reasons [7]. For instance, a developer may want to know where a variable is defined. Feature location [10] and bug localization [11] are examples of semantic information needs. Developers also search over the web for source code examples [7].

Many popular code search tools [12, 13] depend on the user defined terms in code. Table 2.1 shows that popular search engines fail to answer simple queries where there the query terms do not match with code contents. To fix the inefficiencies and shortcomings in such existing tools, a variety of approaches have been proposed. A class of techniques perform query expansion. They enrich the user query with knowledge from sources such as Wordnet thesaurus [14] or developer discussions in a discussion forum [15, 16]. Rahman et al. [15] claim that natural language queries when reformulated with relevant APIs produce much better results. Lu et al. [14] also address the issues related to the quality of the query by expanding it with synonyms from a thesaurus.

Another class of search tools use code snippets as query. They leverage the structure of source code to improve code search. AutoQuery [17] generates Program Dependence Graphs (PDG) from the query code snippets. It uses this dependence information to produce more accurate search results. Wang et al. [18] also show that dependence information in the form of system dependence graphs can be used to improve feature location. Since many code

[1]https://octoverse.github.com/

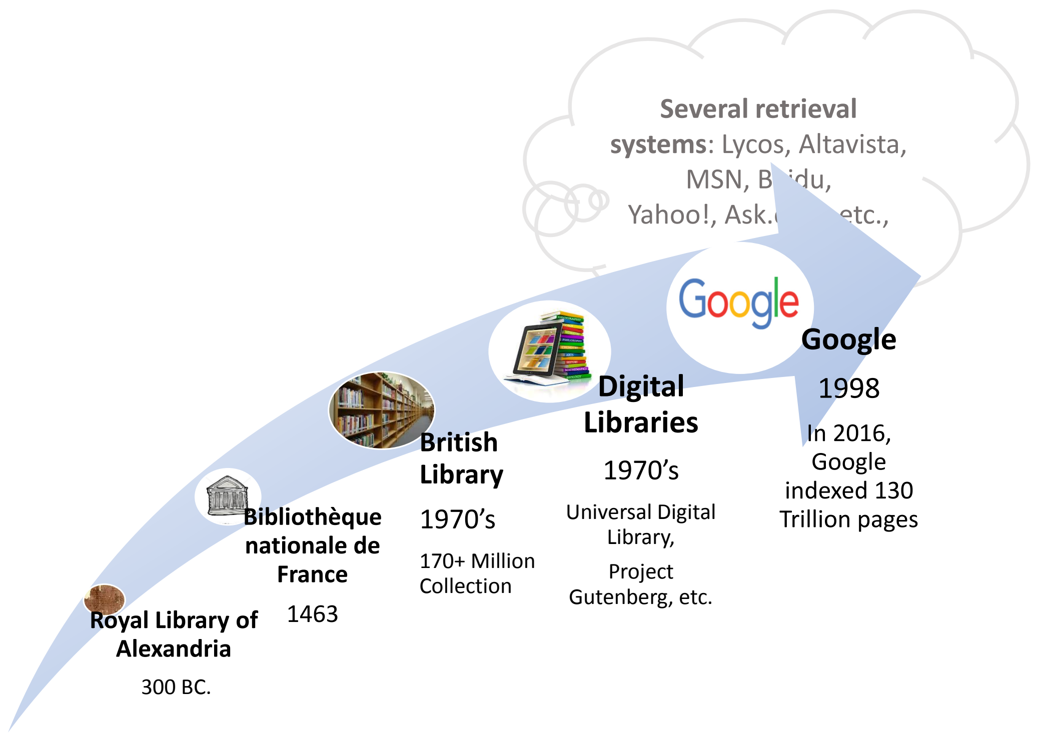

Several retrieval systems: Lycos, Altavista, MSN, Baidu, Yahoo!, Ask.com, etc.,
Google
Google
1998
In 2016, Google indexed 130 Trillion pages
Digital Libraries
1970's
Universal Digital Library,
Project Gutenberg, etc.
British Library
1970's
170+ Million Collection
Bibliothèque nationale de France
1463
Royal Library of Alexandria
300 BC.

Figure 2.1: Growing information needs lead to technology improvement.

Table 2.1: Precision@10 of existing code search engines.

Query	Krugle	open HUB
declare array	10%	0%
concatenate two arrays	20%	0%
check if a String is a numeric type	0%	0%
assign to first element in an array	0%	0%

snippets on the web have parsing and compilation issues, they present challenges in analysis and transformation. Search engines purely based on structural information [19, 20] such as method signatures or call hierarchy have also been proposed. FaCoY [21] is a code-to-code search engine which searches for functionally similar code snippets. Unlike many other search engines which use text query as input, FaCoY accepts source code as input. Yet, FaCoY also formulates a query which is auto-generated from the input code snippet. It uses Q&A posts to find related APIs and tokens to formulate a query. We are also interested in building a code-to-code search engine. Instead of focusing on query formulation and finding similar snippets, we are interested in pairwise code snippet similarity. We are interested in finding variant implementations of input code snippet.

Source code retrieval has attracted the attention of several researchers in the last decade [22, 23, 19, 24, 25]. Literature shows that problems from more than 20 tasks [25] in software engineering have been modeled as retrieval-based problems. Prompter [24] recom-

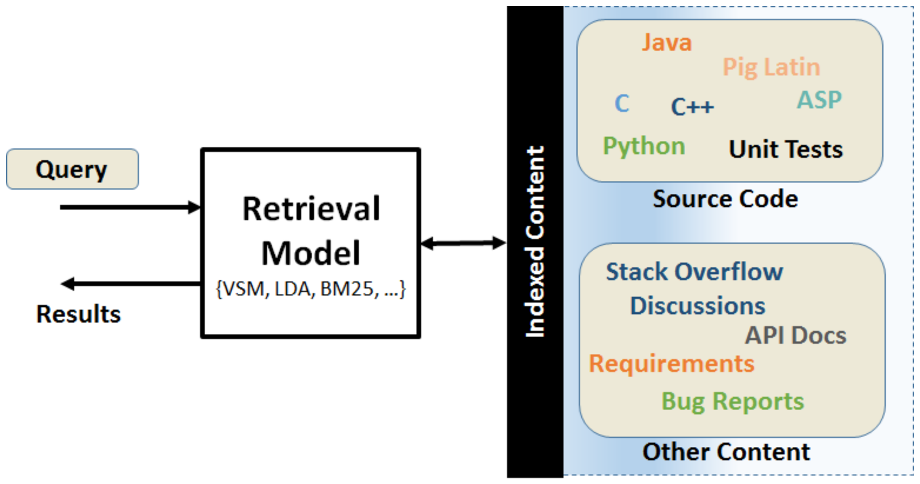


Figure 2.2: A typical IR system which takes queries and returns ranked results. Unstructured content is indexed for faster and efficient retrieval.

mends relevant SO pages for the developer based on the current code context in the IDE. GraPacc [26], a code completion system, is another example for retrieval-based application.

Just like text, source code has also become abundantly available. Defense Advanced Research Projects Agency (DARPA) introduced the term "Big Code" in 2014 to refer to the large corpus of programs, in their article, "MUSE Envisions Mining Big Code" [27]. Platforms such as GitHub[2], bug repositories such as Bugzilla[3], and discussion forums such as SO carry source code snippets in large quantities.

An IR system takes a query and returns a ranked set of results using indexed content. Figure 2.2 shows a typical IR system. Challenges and opportunities in building a retrieval system can classified into two major groups. Firstly, we observe that due to the differences in the nature of text and source code, indexing source code has several challenges. Secondly, the retrieval models that work on text need to be adapted to work with source code. In this Section, we discuss a few related challenges.

2.1.1 Indexing Source Code

Document indexing is the process of mapping terms to documents in the search corpus so that searching is efficient. Unlike text, source code indexing is challenging for the following reasons:

1. *Very few reserved words*: Source code uses very few reserved words in a highly repetitive way. Hence, frequency based techniques on source code may not work without pre-processing.

2. *Tokenization*: Parsing source code requires knowledge of the grammar. Each programming language has its own grammar and hence the process becomes highly platform dependent.

[2]https://github.com/
[3]https://www.bugzilla.org/

3. *Structure*: Structure plays a major role in the semantics. For example, extending an existing class requires the base class to understand the behavior of the extended class.

4. *Identifier Naming*: Identifiers are typically non-dictionary words (such as jSQL, jQuery). Hence, dictionary based disambiguation techniques may not work in such cases.

5. *Ghost Terms*: Many of the queried terms do not exist in code. This is because there exists a gap between natural language (NL) description of a syntactic construct (or a programming concept) such as "array" and its surface form in source code i.e., "[ ]".

A variety of techniques have been proposed to index source code.

2.1.1.1 Indexing the Source Code Vocabulary: Comments and Identifiers

Latent Semantic Indexing (LSI) [28] is an algebraic method to capture the concepts underlying the text. In this method, we start with a term-document matrix constructed by considering each file containing the source code as a document. The terms are usually tokens after pre-processing for stemming and stopwords. We project it to a lower dimensional space using a technique called Singular Value Decomposition (SVD). LSI defines a topic as a probability distribution over the universe of all terms in the documents. This model does not consider the correlation between the terms in the topic. In the context of source code indexing, this model is typically used to capture the comments and user defined terms. Bajracharya et al. [29] introduced a technique called Structural Semantic Indexing (SSI). SSI leverages the idea that a code that uses similar API in a common structure is functionally similar. For instance if two Java methods use the same API in the same order, then these two methods are related even if they have different names. In both these techniques, the main limitation is the inability to index syntactic patterns in source code. They cannot answer queries like "*where is a variable defined?*". Although Program Analysis (PA) techniques are best suited to answer such queries, they do not scale and do not work with partial programs. Partial program analysis (PPA) [30] is one attempt to extend PA in this direction.

2.1.1.2 Indexing Syntactic Structures in Source Code

Apart from the vocabulary, structure plays a key role in distinguishing the way code snippets are written. For instance, a recursive implementation is different from an iterative implementation. Hence, indexing the syntactic structures in source code is important. In this Section, we review two popular techniques that index syntactic patterns in source code.

Code Phrases Nguyen et al. [31] introduced code phrases as a technique to decompose student homework submissions for effective indexing and search. It works in the context where the document space is dense. For the same homework, there are several submissions that vary only moderately from each other. Entire code is converted to AST. Using the various submissions for a programming assignment, common subtrees are identified and one by one, each subtree is removed and kept aside for indexing. The inverted index is constructed by adding one subtree at a time so that there are fewer distinct trees in the index. Identifiers are anonymized. Remaining AST's are hashed so that they can be indexed as strings. This approach might not scale for huge multi-file projects and non-compilable snippets due to the fact that AST can be very large.

Document Fingerprinting Schleimer et al. [32] while implementing MOSS, discuss the technique to efficiently hash code patterns for plagiarism detection. Capturing all possible n-grams of code snippets is perhaps the best way to index syntactical pattern from accuracy perspective. However, such an index will be too large to maintain. Schleimer et al. propose a technique called Winnowing. In winnowing, a fixed number of n-grams are considered in a window and one of them is selected for creating the hash. They show that by using the entropy of strings being hashed, the hash size can be reduced.

These techniques show us how to index code snippets. However, the prior approach depends on snippets to parse into AST successfully. Moreover, ASTs can be extremely large for huge projects thereby causing scalability issues. On the other hand, winnowing works well in a dense code situation. Dense code situation refers to a situation where semantically same code snippets implemented in a structurally similar way are present in abundance. Particularly in academic assignments, a dense code situation can be observed. This might not be the case in several projects where small teams use domain specific languages.

2.1.2 Retrieval Models for Source Code

In the field of Information Retrieval (IR), retrieval models address the task of matching queries to documents, resulting in a ranked list of relevant documents. Various retrieval models have been proposed that use techniques designed for different characteristics of queries and documents. Typically, a retrieval model models the query and the document so as to define a scoring function. The results are ranked based on this scoring. Existing literature represents source code typically in a text format and applies text retrieval models to match them. In this Section, we discuss an algebraic model named Vector Space Model (VSM) [33] which is a popular model for retrieving text as well as source code.

Vector Space Model In VSM, we visualize query and documents as vectors. We match query with the document vector using cosine similarity. Let the query, q be "BITS Pilani". Assume we have two documents, d_1 containing the text, "BITS Pilani Goa Campus" and d_2 be "IIIT Delhi".

If d_1 and d_2 are the only two documents in our index, our dictionary has only six terms. We lay them down in order (need not be alphabetic although that is acceptable too) and collect the frequencies of each term to form a vector. Therefore, if our vectors are made up of frequencies of ("BITS", "Pilani", "Goa", "Campus", "IIIT", "Delhi"), we have, q = (1,1,0,0,0,0), d_1 = (1,1,1,1,0,0), and d_2 = (0,0,0,0,1,1). Then, the cosine similarity between the query and any document d_j is computed as follows:

$$Similarity(q, d_j) = \frac{d_j.q}{||d_j||||q||} \tag{2.1}$$

Applying this measure of similarity in our example, we get, similarity between q and d_1 as 0.71. Similarity of d_2 with q is 0.

Statistical term weighting schemes play an important role in retrieval models because, not all terms are significant in queries and documents. Term Frequency - Inverse Document Frequency (TF-IDF) models solve this problem. A variety of TF-IDF models have been proposed [34, 35, 36].

Augmented TF-IDF Augmented Term Frequency (ATF) is a simple TF-IDF model used for retrieval [37]. ATF $tf(t,p)$ is computed as follows:

$$tf_A(t,p) = \frac{f(t,p)}{1 + max\{f(t,P)\}} \tag{2.2}$$

where $f(t,p)$ represents the raw frequency of term t in post p. Along with term frequency, we need a factor to increase the weight of terms based on the rarity of the term in the entire collection. A standard Inverse Document Frequency (IDF) measure is used in MATF:

$$idf(t) = \log\left(\frac{N+1}{|P(t)|}\right) \tag{2.3}$$

where $|P(t)|$ is the number of SO posts containing the term t, and N is the total number of posts. Given that the SO dump is available, $idf(t)$ can be pre-computed at index-time.

This formulation has the problem that it favors long posts, as the probability of term frequency is proportional to the length of the post. Also, a change in the number of distinct terms in the vocabulary does not affect the TF calculation in this model. Answers in SO posts carry upto 2053 distinct terms. 14 to 66 distinct terms exist in at least 10,000 answers each. Figure 2.3 shows the actual frequencies of answers carrying specific number of distinct terms. It also shows the distribution of SO discussion length in terms of number of words (not necessarily distinct).

Hence, we use the Multi-Aspect Term Frequency (MATF) [38] model. Since we deal with source code in SO, we have adapted MATF in Section 4.2. In Section 4.4, we show that MATF performs better than TF. Here, we give a brief background on MATF.

Multi-Aspect Term Frequency MATF formulation uses a weighted sum of two TF formulations, one which prefers long documents and the other that works better for distinct terms. The MATF score tf_M for a query term t for an SO post p is as shown below:

$$tf_M(t,p) = w \times \frac{tf_R(t,p)}{1 + tf_R(t,p)} + (1-w) \times \frac{tf_L(t,p)}{1 + tf_L(t,p)} \tag{2.4}$$

Here, tf_L refers to the Length Regulated TF (LRTF) component and tf_R refers to RITF. Paik [38] uses $\frac{2}{1+\log_2(1+|Q|)}$ as a value for the aspect weight w. Q represents the set of query terms.

$$tf_R(t,p) = \frac{\log_2(1 + f(t,p))}{\log_2(1 + Avg.tf(p))} \tag{2.5}$$

and:

$$tf_L(t,p) = f(t,p) \times \log_2\left(\frac{1 + Avg.PL}{|p|}\right) \tag{2.6}$$

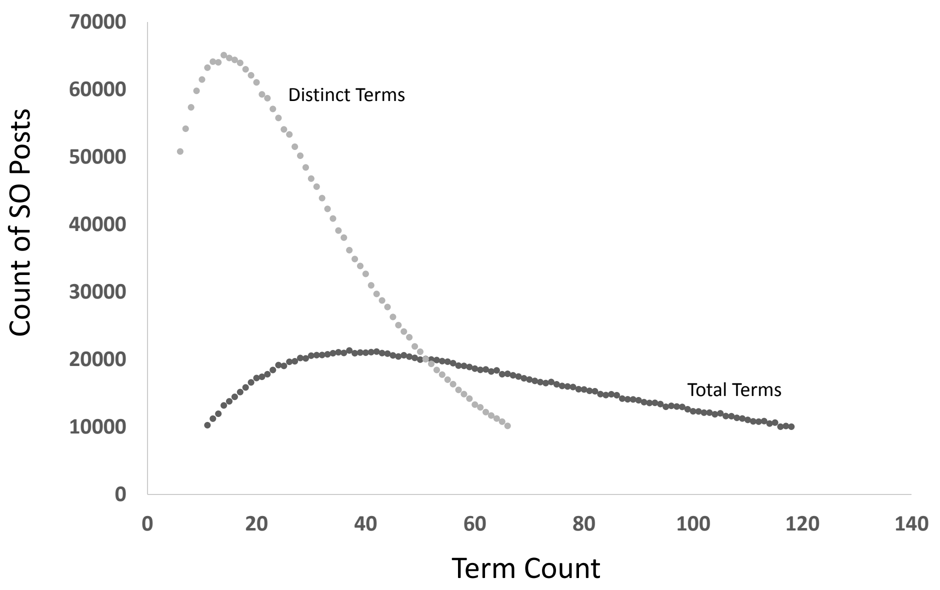


Figure 2.3: Why MATF? SO posts carry developer discussions (answers) that have 14 to 66 distinct terms and 11 to 118 terms overall in them. The distinct terms count range from 0 to 2053. The total terms range from 0 to 8089. This figure considers only those posts where the counts of discussions are above 10,000. Due to the significance that posts carry due to the presence of distinct terms, we use MATF as our retrieval model.

$Avg.PL$ is the average length of the posts in SO, $f(t,p)$ denotes the raw frequency of the term in the post, and $Avg.tf(p)$ is the average term frequency of the document collection (all posts in SO in our case). If there are N posts in SO and $df(q_i)$ is the number of documents containing the query term q_i in query phrase Q, we compute the similarity between query and post by multiplying with the inverse document frequency, as follows:

$$SIM(Q,p) = \sum_{i=1}^{|Q|} tf_M(q_i,p) \times \log_2 \left(\frac{N}{df(q_i)} \right) \tag{2.7}$$

Yang and Fang [39] confirm that MATF gives consistent results throughout several text collections.

2.1.2.1 Limitations of Text Retrieval Models on Source Code

In most existing code search engines [12, 13], queries are expressed as NL text. Many query terms in NL have very different surface forms in source code. For example, “array” in NL is “`[ ]`” in Java. Loop has a pattern similar to “`for` (;;) {}”. Thus, the query terms that are missing in source code cause the search to fail. Few search engines [40] allow structural queries. Yet, they cannot answer queries such as “Where is an integer array declared” or “Get me methods that implement factorial”. Code contexts are hard to extract. Many tools such as code completion, example recommendation, API Usage recommendations need to understand the current code in IDE and the context in which recommendations are sought.

Typical ideas such as co-occurrence [41] analysis do not work well to capture such contexts. Moreover, a variety of programming languages pose challenges in extracting the current code context.

In summary, gaps exist between the source code representations and the assumptions of existing retrieval models. This presents interesting space for research. In this thesis, we leverage big code opportunities, and investigate modeling source code and designing retrieval models to support retrieval-based applications.

Chapter 3

Code Variants

There are often multiple ways to implement the same requirement in source code. Different implementation choices can result in code snippets that are functionally similar. In the existing literature, these code snippets have been defined in multiple ways such as code clones, examples and simions. Variants are fundamentally different from these known types of code snippets. Currently, there is a lack of a consistent and unambiguous definition of code variants based on their intended usage. Code variants are a specific type of code snippets that differ from each other by at least one desired property within a given code context. We distinguish code variants from other types of semantically similar snippets in source code, and demonstrate the significant role that they play. We observe that about 25% to 40% of developer discussions in a set of 15 open source projects are about variants. We characterize variants based on code context and desired properties. We study if developers seek variants and then report the effect of variants on developers' seeking behavior. Our findings call for building automated tools to search, compare, and synthesize variants.

3.1 Introduction

Programming is a creative activity with many different ways to implement the same requirement. Developers often have to reason among the benefits and drawbacks of different implementation choices before making a selection. When evaluating the choices among different code snippets, developers have to consider properties such as, differences in the speed or complexity of computation, the style of coding, the library used, or licensing requirements.

Research has identified different situations where code snippets can be similar, from being exactly the same to being similar in some dimension. Figure 3.1 presents the different types of code similarity referred in software engineering research. Code might be exactly the same (and repeated) at the token-level and the line level as demonstrated by the work on naturalness of software [42]. Idioms, clones, simions, and code variants can span multiple lines of code that are similar. Similarity can also be at a higher granularity, occurring beyond individual programs and encompassing applications and products. Our work focuses on (similarity of) code snippets.

We call those code snippets as variants, when they have the same behavior under a given code context, but differ on other properties that make one snippet a better fit than

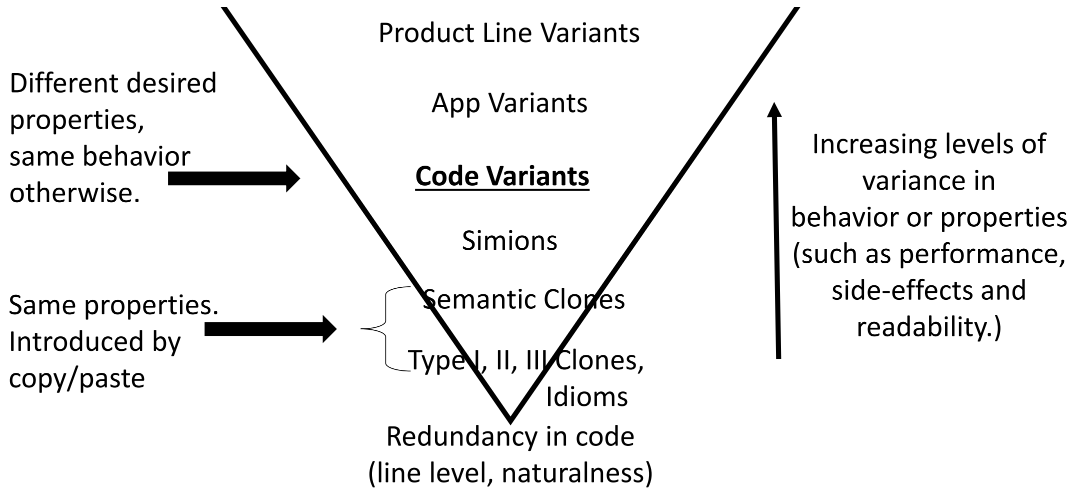

Product Line Variants
App Variants
Code Variants
Simions
Semantic Clones
Type I, II, III Clones, Idioms
Redundancy in code (line level, naturalness)
Different desired properties, same behavior otherwise.
Same properties. Introduced by copy/paste
Increasing levels of variance in behavior or properties (such as performance, side-effects and readability.)

Figure 3.1: Reuse in source code happens at different levels. This work focuses on code variants.

Table 3.1: Code variants are discussed in defect descriptions of Apache Math, a popular open source project.

Project	**Discussion**
Apache Math [Bug# MATH-901]	One of the reasons this **variant** is faster is because it is less accurate, which may not be acceptable for commons-math.
Apache Math [Bug# MATH-1293]	While the jury is still out, I made another **variant** of the patch ...

the other. Developers are known to remove clones and simions [43] to promote reuse and aid maintenance. On the other hand, variants are desired and sought by the developers as replacement to existing code without which the existing snippet misses desired properties.

We do not yet have consistent terminology or rigorous definition for these types of replaceable code snippets that are “a better fit” than the existing snippet. Developers loosely use the term “variants” (see Table 3.1) to refer to alternative implementations of a code snippet. We find that developers evaluate variants while discussing bugs. They use discussion forums while searching for better implementation choices. In the absence of clear definitions of the dimensions along which variants are similar or differ, automated tools cannot support developers’ decision making. To the best of our knowledge, this is the first study conducted to understand variants (that should be brought into the codebase) and their characteristics. Our study answers the following research questions:

RQ1: What are code variants? Are they different from known forms of code snippets such as clones, examples, idioms and simions?

RQ2: What are the aspects of variants that developers consider while describing variants?

RQ3: Do developers seek variants? Is there a difference in their seeking behavior based on the nature of variants or on programming language?

Our study results indicate that variants form an important category of code snippets. A better understanding of variants will open up a new field of research, benefiting tool builders and researchers, which we discuss as implications.

3.2 Background and Related Work

Code variants are closely related to two major types of code snippets: 1) Semantically similar code snippets (clones, idioms and simions), and 2) Code examples. Each of these types has similarities and dissimilarities with variants.

3.2.1 Background on the Types of code snippets

Here, we briefly give a background on popular forms of redundancies in code, namely clones, idioms, simions and examples. In Section 3.4, we define and distinguish variants from all these types.

Type I, II, and III Clones Clones were originally defined as redundant snippets introduced due to a copy and paste activity [3]. Syntactic clones [44, 45] are of three types. Type-1 clones are exact copies. Type-2 clones are copies where only the identifier names and variable types are changed. They are otherwise structurally similar. There is no single accepted definition of Type-3 clones [44, 45]. One definition is based on the Levenshtein distance between the pair of snippets which quantifies the minimum number of additions and deletions of tokens to transform one snippet to other.

Semantic Clones Semantic clones appear in different varieties, such as wide-miss clones [4], interleaved clones [5], and high-level concept clones [4]. Gabel et al.'s [5] definition of semantic clones is as follows: *Two disjoint, possibly non-contiguous sequences of program syntax $S1$ and $S2$ are semantic code clones if and only if $S1$ and $S2$ are syntactic code clones or $\rho(S1)$ is isomorphic to $\rho(S2)$*. Here, ρ is a Program Dependence Graph (PDG) based transformation function. PDG captures control and data dependency in code snippets and abstracts away other syntactic details. Elva and Leavens [46] define semantic clones as functionally identical code fragments. Ira Baxter defines clones as segments of code that are similar according to some (typically lexical) definition of similarity [44].

Idioms Keivanloo et al. [45] and Juergens et al. [43] indicate that code similarities may exist beyond these clone types. Perlis [47] introduces idioms as *language constructs characterized by frequency of occurrence, unity of purpose, ease of recognition, and composability of use*. Allamanis and Sutton [48] define a code idiom as "*a syntactic fragment that recurs across software projects and serves a single semantic purpose*". For example, `for(int i=0;i<n;i++) { ... }`. They claim that programmers use the term idiomatic to refer to a code snippet that is used repetitively.

Simions Juergens et al. [43] call the code snippets that are behaviorally similar as *Simions*. Simions need not originate from copy and paste activity. They argue that simions cause maintenance issues and therefore treat them as entities that programmers would like to clean up in source code. They also claim that existing clone detection tools find less than 1% of simions.

Table 3.2: Fundamental differences exist between semantic clones, code examples and code variants.

	Semantic Clones	**Code Examples**	**Code Variants**
Definition	Code snippets with no difference in properties of interest within the given code context. Therefore, one snippet can replace the other.	Code snippets with an instructive property against an information need.	Code variants represent alternative implementations suitable for a specific code context in which one variant must have some desired properties over the other.
Differences	Clones are necessarily semantically similar and have no differences in desired properties. Hence, these snippets are redundant. Some amount of structural similarity is also assumed in cases where PDG based definitions are followed.	Neither semantic nor syntactic similarity warranted. Provides instructive value as in the usage of API or how to implement, and so on.	Semantically similar but has different desired properties.
Example	Two sorting implementations of same worst-case complexity where that is the only quality that matters to developers.	Any API usage tutorial. For example, in Java, the code snippets describing the usage of Arrays.sort feature.	Various sorting implementations with different time complexities are variants, if speed is the only desired property.

Code Examples Developers seldom read the entire documentation before they start. They learn from code snippets on the web or other projects [49, 50]. *Code examples are small source code fragments whose purpose is to illustrate how a programming language construct, an API, or a specific function or method works* [51]. Examples play a significant role in comprehension, reuse, and bug-fixing [52]. As a result, several researchers have explored locating [48, 53], selecting [54] and analyzing [52] examples.

Table 3.2 summarizes the differences between semantic clones, code examples and variants.

3.2.2 Redundancies in source code

Hindle et al. [42] observe that source code being a human product is repetitive. They show that repetitions occur at n-gram level where n can be as low as 3. Juergens et al. [43] claim that semantically similar code taken from various sources can be syntactically heterogeneous. These snippets as we have discussed so far, appear with various names. In another study [3], they also report that inconsistent changes to clones lead to maintenance issues which happens to be the key concern while dealing with clones. We leverage these studies but find that a definition of code variants is necessary. That is the focus of our work.

3.2.3 Variants: A missing link

Use of the term “variant” is quite popular in the development community. In the Gabel’s dataset, originally used for clone detection, a search for the term “variant” leads to a large number of defect reports (GIMP: 109 occurrences, GTK: 324, MySQL: 166, Postgresql: 262, and Linux Kernel: 286). Not only developers but literature too provides several evidences of the use of the term “variant” [55, 56]. The term “Variants” occur in at least three major forms: 1) Code variants (focus of this chapter), 2) Program or product variants (as in product lines and application variants) [57], and 3) Configuration variants [56] (as in tuning a product or product configuration).

3.3 Research Methodology

Our goal is to investigate to what extent developers discuss code variants. Therefore, we analyzed the discussions that have taken place in issue tracking repositories and SO posts. We used a mixed methods approach, where we combined quantitative and qualitative analysis of the developer discussions. For the qualitative analysis, we followed a similar approach as Ma et al. [58] and manually analyzed bug reports, SO posts, and research literature to define, identify, and classify code variants. Three graduate students with two years of Java coding experience each, were responsible for the qualitative coding, which used negotiated agreement to reach consensus (100% agreement), after which each student individually coded their assignments.

We extracted 1500 bug discussions, 100 each from 15 open source projects (Table 3.3), belonging to five different programming languages.

Programming Language Selection

When selecting the projects, we wanted to include a variety of programming languages, domains of use, and project sizes. We selected five languages that are popular in the research community of clones, idioms and examples. These languages were Java, C, Python, Javascript, and Ruby. This set of five languages provided the diversity in terms of programming language characteristics such as, being object orientated or procedural, strongly or weakly typed, and compiled or interpreted. We selected three projects for each language.

Project Selection

Our selection of projects was guided by existing relevant research literature. We selected the Java projects –Atmosphere and Hibernate– as they were evaluated by Allamanis et al. [59] in their study of idioms. To add diversity in the project domains we also included Apache Math, which is an algorithmically rich library written in Java. The three C projects: GIMP, GTK+ and MySQL, were used by Gabel et al. [5] in their work on semantic clones. The Python projects: Plone, SCons, and Zope, were used by Roy et al. [60] in their study of code

Table 3.3: Projects used to characterize variants.

#	Project	PL	LoC	Domain
1	Apache Math	Java	375K	Mathematics
2	Atmosphere	Java	68K	Client-Server
3	Hibernate ORM	Java	930K	Domain Model Persistence
4	Gimp	C	780K	Image Manipulation
5	GTK+	C	880K	UI Widget Toolkit
6	MySQL	C	1130K	Database
7	Plone	Python	74K	Content Management
8	SCons	Python	228K	Build Tool
9	Zope	Python	272K	Web Application Server
10	Bootstrap	Javascript	37K	Mobile First Projects
11	Foundation	Javascript	53K	Web Front End
12	Jquery	Javascript	46K	Client-side Scripting
13	Rails	Ruby	224K	Web Applications
14	Fastlane	Ruby	70K	Releasing Mobile Apps
15	Huginn	Ruby	33K	Task Agent Builder

clones. We chose the JavaScript projects: Bootstrap, Foundation and jQuery, as they were also used by Cheung et al. [61] in their study on code clones. We pick the Ruby projects: Rails, Fastlane and Huginn, to add diversity to our programming language selection. These Ruby projects are among the top 10 most trending (highest stars) Ruby projects on Github. The projects in our dataset vary from 33K (Huginn) to 1130K (MySQL) lines of code, and include disparate domains such as, mathematics, databases and editors.

Defect Discussion Labeling

We picked 100 random defect discussions from the issue tracking repositories of each project for analysis. The issues were selected by using a script, which took the open defect Id range as input, and selected 100 random defect Ids. Three annotators with a programming experience of at least two years in all the five languages, shared the task of analyzing these defect discussions.

Annotators used negotiated agreement when labeling the discussions. Each annotator individually analyzed the first 30 discussions in the first project assigned, to identify if the discussions were about variants and if so, the annotator labeled the other characteristics (context type and desired property as explained in Section 3.4). All annotators then discussed their annotations with one another, and where needed refined the definitions and the analysis checklist. They continued this process until they attained consensus (100% agreement). Then, they worked individually again to complete the annotations for that project and the other projects assigned to them.

Thus, we arrive at a dataset that includes 1500 discussions, with 300 discussions per language.

3.4 RQ1: What are Code Variants? Are they different from the known code snippet types?

As a first step towards creating a consistent terminology of whether a given snippet of code is a code variant, we propose a formal definition of code variants. Three developers with at least two years of Java coding experience manually inspected the code snippets discussed in our issue repository dataset. They annotated each code snippet with its type (such as clone, variant, simion) based on the definitions in Section 3.2. We then analyzed the code snippets that were not clones or simions and arrived at the following formal definition of code variants that we follow throughout this dissertation. We revised the definition until we obtained 100% consensus on whether a discussed code snippet is a variant.

A code variant represents an alternative implementation for a given code snippet under a specific context in which one of the two implementation choices must score better on at least one desired property over the other. More formally, we define it is follows:

Definition 1. *Let P be the set of n desired properties $\{p_1, p_2, ..., p_n\}$. Let $score_{p_i}(\nu)$ be a function computing the strength of code snippet ν over any property p_i. Code snippets ν_1 and ν_2 are **Variants** if there is at least one property of interest by which ν_1 is better than ν_2, or ν_2 is better than ν_1 in the given code context. Both ν_1 and ν_2 should be acceptable in the current code context i.e., $\exists_i$ $score_{p_i}(\nu_1) \neq score_{p_i}(\nu_2)$.*

Note that, in a game theory parlance, the definition of variants requires them to be pareto optimal over $score_{p_i}(\nu)$. This definition also emphasizes on *code context* and *desired properties* of code snippets. We adapt Kirke's [62] definition of method context to define code context as follows:

Definition 2. ***Code Context*** *of a code snippet describes the "fit" of the snippet inside the larger project. It captures the intent, dependencies of the surrounding code, input to the snippet, the output from the snippet, and the states in which the system may get into.*

In a variant pair, one variant is said to be *preferred* over the other if it has at least one *desired property* when compared with the other within the given context. For brevity in the rest of the dissertation, we use "context" to refer to code context, and "properties" to refer to desired properties. We define desired properties as follows:

Definition 3. *Multiple implementation choices may satisfy the requirements in the given code context. Yet, each implementation choice (a code snippet) differs from one another in ways that could be either functional or para-functional or both. These qualities that serve as differentiators are referred to as **Desired Properties**.*

Variants are neither clones, nor simions, nor idioms Code variants are similar to clones in the sense that both are functionally similar set of code snippets. Variants differ from clones for the reasons of purpose and properties in the code context. Variants are always discussed with the intent of *bringing in* code snippets with desired properties. Clones are discussed in the context of refactoring. Developers *clean up* clones to promote reuse. Existence of a clone is considered as a bad smell. There is no difference between the desired properties present in the clone instances.

Gabel's definition of clone does not capture these aspects of semantic clones. We suggest the following definition:

Definition 4. *Code snippets ν_1 and ν_2 are **Clones** if neither ν_1 nor ν_2 score over each other on any property of interest and thus ν_1 and ν_2 can replace each other in the given code context. In other words, $\forall_i\ score_{p_i}(\nu_1) = score_{p_i}(\nu_2)$.*

This definition does not depend on structural similarity at all, and instead focuses on desired properties in a code context. This emphasis helps us differentiate variants not only from clones, but also from simions and idioms as well. This definition applies to all types of clones including semantic clones. Individual types put forth further restrictions. Since the types of clones is not the focus of this chapter, we do not go deeper to define them individually. Simions and idioms are semantically similar irrespective of the code context.

Variants are not examples Unlike other types, code examples need not be always similar in behavior. Behavioral similarity follows from Definition 1 as a prerequisite for variants as both snippets must be acceptable in the given code context. For instance, examples could be instructive to explain API usage in a variety of functionally different snippets. Thus, an example mining tool cannot be a variant mining tool. This necessitates a synthesis step to make the example fit for use in the given context.

Variants are not bug-fixes or enhancements Let ν_2 be an enhancement sought over ν_1. Even though it seems that an enhancement may add some desired property to the existing code, we observe that the intent has changed. Moreover, the "desired" property has now become a "required" property. As an example, ν_1 and ν_2 may have been (and need not always be) variants in the earlier code context when ν_1 was under development; however, in the new context, ν_2 alone is acceptable and ν_1 does not fit. Same argument applies to bug-fixes as well. A buggy-snippet does not meet the expectations of the code context, and hence is no more a candidate for being a variant.

3.5 RQ2: What are the aspects of variants that developers consider while describing variants?

Developers use code context and desired properties to articulate about code variants (as discussed in Section 3.4). Here we define each of these aspects by surveying relevant research literature and then evaluate our definitions by analyzing the developer discussions.

3.5.1 Code Context

While desired properties distinguish variants, code context relates them together. As discussed in Section 3.4, code context description comprises of one or more of the following: intent, dependencies, input/output and state. We arrived at this taxonomy based on a literature survey of 11 related research papers [63, 64, 24, 65, 2, 66, 67, 68, 69, 70, 71] and our experiments on 15 open source projects (See Section 3.6).

Intent Programs are products of human desire to solve specific problems or accomplish well-defined tasks. Hence, an understanding of the problem being solved plays an important part in recommending variants that "fit" the purpose. Purpose includes functional and para-functional requirements. For example, "computing factorial", "implementing little endian algorithm" are examples of intent. Intent specification is a hard problem [63] which goes beyond just naming and describing the problem using natural language phrases. Nguyen et al. [64] relate execution context and intent. Their thesis is that the intent can be captured using API usage patterns in the code.

Dependencies Often, implementation is constrained to a specific programming language, certain pre-built libraries, or components. Search for variants must honor these constraints. Constraints may also include structural elements such as methods or classes as in Java. We refer to such constraints as dependencies. For example, a REST API for financial data may be provided by multiple providers which become variant choices. In this context, we assume that non-REST APIs are not sought by the developers. Robillard [65] claim that neglecting such dependencies may lead to *low-quality modifications*. They discuss structural dependencies in the scope of *program elements* and mention *methods* and *fields* as examples.

Input/Output (IO) Input and output examples are used as context to search [2] and synthesize [66] source code. Nix argues that the problem of synthesizing expressions mapping given a set of inputs to the given set of outputs (in the sub-context of repetitive text editing) is NP-Hard. Programming-by-Example community shows steps taken in this direction with string transformation [67]. In summary, a decade old research in this area has produced solutions for text editing and spreadsheet processing; however, synthesizing large sized programs remains a challenge [68].

State Often, developers complain of a specific state that the system gets into. For example, in Zope[1], a developer states, *"For huge transactions ZEO spends a long time (in the order of minutes) in the call to "vote". This makes it irresponsive for other request..."*. Current context of the code under execution includes the snapshot of its variables, the line under execution, and the resources available at that time for the program [69]. This definition of context is used heavily in debugging [69], program repair [70], and real-time updates [71] to software systems. These systems use a variety of techniques such as automata and logic for capturing and representing the context.

Next, we evaluate whether our definition of context and its types is sufficient to cover all the variants in our dataset. We also analyze if certain context types are more dominant than others in our dataset.

How are variants distributed across the context types?

Developers need to evaluate the fit between the code context of the variant and their program. Therefore, it is important to understand the extent to which code context plays a part in developers' discussions. Table 3.6 (a) provides samples of discussions regarding the four

[1]https://bugs.launchpad.net/zodb/+bug/143274

Table 3.4: Code contexts are primarily described using one or more of these four types. We map defect descriptions containing variants to these types.

#	**Project**	**Intent**	**Dep.**	**IO**	**State**
1	Apache Math (Java)	33	11	2	0
2	Atmosphere (Java)	24	12	2	1
3	Hibernate ORM (Java)	18	9	1	3
4	Gimp (C)	16	6	3	10
5	GTK+ (C)	23	4	4	8
6	MySQL (C)	21	3	9	9
7	Plone (Python)	19	7	8	6
8	SCons (Python)	17	10	0	3
9	Zope (Python)	20	7	4	10
10	Bootstrap (Javascript)	16	2	14	7
11	Foundation (Javascript)	14	5	13	16
12	Jquery (Javascript)	12	4	8	13
13	Rails (Ruby)	30	1	1	1
14	Fastlane (Ruby)	24	1	4	2
15	Huginn (Ruby)	14	8	13	11
	Total	**303**	**83**	**90**	**100**

context types. For example, row 1 in Table 3.6 (a) shows a snippet from a discussion in Apache Math (Defect: 785) about variants with an intent to compute continued fraction. The discussion in Atmosphere (Defect: 2037, row 2, Table 3.6 (a)) is about the common state of *disconnect* method while closing different browsers.

We were successful in classifying all the contexts in the variants into these four code context types. Two graduate students separately classified the context in all the 441 variants, and there were no disagreements about the context boundaries. Table 3.6 (a) shows a sample discussion for each context type.

Observation

Our empirical findings confirm that the four context types are sufficient to cover all the variants discussed in our dataset.

Table 3.4 presents the breakdown of the variant discussions pertaining to context types. We observe that intent dominates in this list amounting to **52.6%** (303 out of 576 contexts) across the variant discussions, where the discussions were about the underlying functionality of code variants. Dependency, IO and State covered 14.4%, 15.6% and 17.4% of contexts observed in discussions, respectively.

These findings indicate that developers deliberate about the context in which variants are to be used, and could benefit from automated tool support.

Observation

Intent was the most common context covering 52.6% of the code contexts in variant discussions.

Table 3.5: Examples of developer discussions taken from Eclipse project describing the desired properties in code variants.

Property	Defect Id	Developer Discussion
Algorithmic	384730	There are already some implementation of this algorithm. However, most of them are pretty complex and slow. I would like to contribute a smaller and simpler version compatible with the ZEST layout engine.
RO	293637	Ribbon must be licensed by each adopter. If Eclipse will provide Ribbon, than every RCP application with Ribbon must be licensed. This violates EPL.
Pure-Diction	196585	It is better to use the setter methods on the model classes (e.g. TracWikiPageVersion) than having constructors with many parameters. That way the order of the parameters does not get mixed up and the code is easier to refactor and to read.
Mechanics	338065	Our coding conventions currently demand to declare all method parameters as final in order to prevent parameter assignments. Meanwhile, parameter assignments can effectively be revealed by the Eclipse tooling and by tools like FindBugs on the CI server.

3.5.2 Desired Properties

As discussed in Definition 3, desired properties distinguish variants. Desired properties in a variant can be classified into broadly three groups: a) Algorithmic, b) Resource-Oriented and c) Diction.

Algorithmic Properties Algorithms play a significant part in computation and their properties are well studied [72]. Developers seek efficient algorithms to make their code score on para-functional attributes such as, security [73], accuracy [74], readability [51], and scalability [5]. Sridhara et al. [75] discuss the importance of identifying high-level algorithmic steps in source code. Patterns, signatures and structures are limited in their ability to detect algorithms in source code [76]. Many reuse techniques [29, 77] that work at function level focus on semantic similarity and ignore the variability across variants. Mishne et al. [76] extract concept graphs to represent algorithmic information.

Resource-oriented (RO) Properties For reasons such as licensing [78, 79], certain libraries, components, sub-systems, interfaces, and services are considered better or relevant. This property has nothing to do with the syntax or semantics of the code snippet. Instead, it is about the extraneous (non-code) elements associated with the snippet, such as the legal constraints, and trust factors. Long [80] observes that many third-party libraries are no longer actively maintained. He calls this the *used car fiasco.* He brings up more issues in reuse, such as *One size fits all* and *Of course it's reusable.* Moreno et al. [51] discuss the effort to reuse the code snippet.

Diction Properties Diction refers to the *style of speaking or writing as dependent upon choice of words*[2]. Some developers may prefer `for` over `while` to code a loop. Resulting

[2]http://www.dictionary.com/browse/diction

code is semantically the same. Naming conventions may contribute to the ranking of one variant over the other [81]. We call such variants as diction variants. Diction variants cover all non-algorithmic and non-resource-oriented properties, such as patterns, refactoring needs, conventions and style. Often, programming language libraries give multiple ways to implement the same functionality within the same resource and algorithmic constraints. Syntactic sugars [82] are classic examples for this type of variants.

Diction variants can be further classified into two types: *Pure-Diction* and *Mechanics*. Pure-Diction refer to those variants that differ only in the style of writing by way of using different syntactic constructs. The loop elements such as `for` and `while` belong to this type. Another class of Diction is made of those variants that have structural differences with hidden properties which may case side-effects, although such side-effects are not expected to show any behavioral differences for the current code context. An example is the use of a different set of parameters to a method in Java.

Certain syntactic choices have distinguished benefit over the other. A recursive version of factorial is rarely used in practical scenarios. A memoization approach avoids recomputation and is desired especially when large inputs values are bounded so that a four byte variable such as Java *int* can hold the result. Yet, the role of diction variants have been largely ignored by the research community. In academic context, most plagiarism tools depend on these differences to avoid marking student works as duplicate.

Absence of one or more of these properties leads to low-quality code snippet for which developers seek replacement. This absence may introduce faults, bad smells or sub-optimal code. Table 3.5 shows real developer discussions from Eclipse and HTTPClient projects. We have mapped these discussions to one of the three types of properties discussed. Next, we report our empirical findings from this exercise.

How are variants distributed across the types of desired properties?

Desired properties distinguish one variant from another. Table 3.6 provides samples of discussions regarding the four different types of desired properties. For example, row 1 in Table 3.6 shows a snippet of a variant discussions in MySQL (Defect: 42948) that relates to the underlying algorithm (Algorithmic property), where developers deliberate on the performance issues. As another example, the discussion in Atmosphere (Defect: 888, row 2 in Table 3.6) is about the management of execution threads (RO) for an application.

We were interested to see if the set of desired properties (Algorithmic, RO and Diction) that we define in Section 3.4 are adequate in categorizing the different variant discussions in our dataset. We mapped each variant discussion with at least one of the desired properties. In no case, did we need a definition of an additional desired property.

Observation

Our empirical findings validate that the four types of desired properties are adequate to cover all the variants discussed in the entire dataset.

Next, we investigate the distribution of variants across these different properties. Table 3.8 lists and Table 3.9 summarizes the distribution of the variant discussions across the different desired properties. We find that Algorithmic variants dominated in MySQL accounting for 64.3% (18 out of 28: Table 3.8) of discussions. MySQL is a project about database management, and a substantial set of discussions were about the algorithm such as, optimizing the queries to scale over large database with discussions on indexing and caching techniques. Overall 41.07% (175 out of 426: Table 3.9) of discussions were of Algorithmic type.

We find that RO variants, in our dataset, are discussed only when there is a concern regarding issues such as, licensing, library compatibility, or coding conventions. RO variants dominated in the Fastlane project (a tool to release mobile applications) accounting for 39.3% (11 out of 28: Table 3.8) of discussions. However, overall only 6.6% of total discussions were about RO variants. One reason for fewer discussions about RO could be that issues regarding licensing, library compatibility and coding conventions are functions of attributes such as, domain and developer competence. In contrast, our study focuses on language, context, properties, and variant types.

Diction was the dominant variant property accounting for **43.4%** (185 out of 426: Table 3.9) of discussions. Out of the 185 diction variants overall, we found 39.5% (73 out of 185: D_p in Table 3.9) Pure-Diction variants, and the rest 60.5% (112 out of 185: D_m in Table 3.9) were of Mechanics type. Discussions about diction variants were typically regarding better coding conventions and styles. Diction variants in large numbers indicate that developers care for style. Table 3.6 (b) shows such sample discussions about style in the last two examples (in Plone and GIMP).

Observation

43.4% of variant discussions talked about Diction which was found to be the dominant desired property.

Table 3.6: Discussions (with referenced defect from the issue tracker) from the dataset capturing the following: a) the desired properties, and b) the code contexts across variants. (S# captures the key statements from the discussion)

(a) Context	Excerpts from the Discussion
Intent `MATH-785`	**S1:**The ContinuedFraction calculation can underflow in the evaluate method, similar to the overflow case already dealt with. **S2:**The evaluation of the continued fraction has been changed to the modified Lentz-Thompson algorithm which does not suffer from underflow/overflow problems as the original implementation. **Essence of the Discussion:** Variants have a common intent here which is to compute continued fraction.

Table 3.6: Discussions (with referenced defect from the issue tracker) from the dataset capturing the following: a) the desired properties, and b) the code contexts across variants. (S# captures the key statements from the discussion)

State `Atmosphere-2037`	**S1:**the _disconnect method is not being called for Android Chrome when closing the browser though the android apps panel. Only in case of changing the URL on the browser, that method is called. **S2:**It works with Firefox desktop, Firefox mobile, chrome desktop, etc. **Essence of the Discussion:** State of disconnect method invocation on closing different browsers being discussed.
Input/Output `MATH-1143`	**S1:**A DerivativeStructure and UnivariateDifferentiableFunction are great tools if one needs to investigate the whole function but are not convenient if one just needs derivative in a given point. **S2:**Give the derivatives in the “natural” order, which is in increasing order when you have one parameter and high order derivatives, and in parameters order when you have only first order derivatives for all parameters. **Essence of the Discussion:** Common output of derivative, across all variants.
Dependency `MATH-1098`	**S1:**As we will certainly not add a dependency to another library, we could start with our own set of annotations **S2:**I recommend using @Retention(RetentionPolicy.SOURCE) so the annotations don’t add to the jar file size **Essence of the Discussion:** Two variants of a function where both do not have a dependency on another library.
(b) Desired Prop.	**Excerpts from the Discussion**
Algorithmic `MySQL-42948`	**S1:**Each of the views on which I was doing a join was doing a full table scan of 1.2 million records instead of using the “CustomerID” index **S2:**To restore normal performance, I had to write stored procedures to create temporary tables instead of using views **Essence of the Discussion:** Performance being discussed across variants to execute a query over a large data table.
RO `Atmosphere-888`	**S1:**Add timeout support for WebSocket to prevent thread waiting indefinitely **S2:**configure the buffer size as well **Essence of the Discussion:** A variant to better manage threads(resources).

Table 3.6: Discussions (with referenced defect from the issue tracker) from the dataset capturing the following: a) the desired properties, and b) the code contexts across variants. (S# captures the key statements from the discussion)

Pure-Diction `Plone-732`	**S1:**The toolbar displays the "xx days ago" information during the loading of a page as ISO date time string and then turns it into "xx days ago" after the complete loading of the HTML. This confuses the eye especially when the page is loaded over a slow line or takes some time for rendering. The toolbar should only display "xx days ago" **S2:**Perhaps the toolbar or this particular toolbar items should be hidden by default and made visible after moment.js **Essence of the Discussion:** Alter the content displayed.
Mechanics `GIMP-737778`	**S1:**Currently some GIMP editing operations are hard-coded to use parameters specific to sRGB. To allow correct editing in other RGB working spaces, the hard-coded sRGB parameters must be replaced with parameters retrieved from the image's actual RGB working space. **S2:**I personally think native support for color spaces other than sRGB is a great thing **Essence of the Discussion:** Allow passing as parameters instead of using hard-coded values.

3.6 RQ3: Do Developers Seek Variants?

We categorized discussions as about variants if they discussed implementation choices. Our dataset had on average **28.4%** (426 out of 1500) discussions (see Total in Table 3.9) where developers discussed about variants. The number of variant discussions (see T_v in Table 3.8) ranged from **25%** (in Plone) to **40%** (in Apache Math).

Observation

In 25% to 40% of defect discussions, developers actively seek and compare variants.

While annotating discussions, we observed that the developer behavior was not same while seeking any variant. This led us to the classification of variants based on the effort required by the developers access variants. In this Section, we discuss this nature of variants. Further, we analyze if the programming language affects variant seeking behavior.

3.6.1 Nature of Variants

Developers need to decide if a particular variant is better or worse (simple) or even incomparable (complex) to the rest. To this end, we show that variants exhibit a strict partial

Table 3.7: Examples of developer discussions describing the variant types.(S# captures key statements from the discussion.)

Type	**Developer Discussion**
Simple GTK+ 109292	**S1:**The function does a linear search in the array instead of a binary search. Also, even when the model is caching iters, it doesn't use that information but slowly converts to a path and back. **S2:**I've been using this patch and it's an amazing improvement on the previous implementation, which was so slow **Essence:** Discussion on faster algorithm, accepted by all(indicating no tradeoff likely).
Complex Huginn 1940	**S1:**On a very low-volume instance, at some point, events simply stop to be processed. Never delete the event with the highest ID. Put a warning next to the field where you choose the retention period Avoid/discourage the use of InnoDB for the events table. **S2:**In general, InnoDB is more reliable than MyISAM, and has better transaction support. It'd be a shame to lose that. **Essence:** Discussion is over avoiding a storage engine. Some developers are reluctant due to reliable support.

Table 3.8: Volume of variant discussions in open source projects depends on the project domain. T_v, *Alg*, *RO*, D_p, D_m are the counts of: total, algorithmic, RO and pure-diction and mechanics variants respectively.

Project	T_v	**Simple**				**Complex**			
		Alg	*RO*	D_p	D_m	*Alg*	*RO*	D_p	D_m
Apache Math	40	5	1	7	4	12	7	4	9
Atmosphere	28	2	7	2	6	6	3	0	5
Hibernate	25	5	3	4	7	2	5	2	2
GIMP	34	11	2	2	5	7	4	2	3
GTK+	29	6	3	3	4	8	3	2	2
MySQL	28	12	1	4	3	6	4	1	0
Plone	25	2	3	7	2	4	3	3	2
Zope	26	10	2	2	1	6	3	1	2
SCons	25	8	0	1	5	3	2	2	5
Bootstrap	25	3	1	4	5	6	3	2	2
Foundation	27	6	3	3	4	6	3	0	3
Jquery	26	3	2	1	4	9	2	0	5
Rails	33	5	6	4	8	5	3	2	0
Fastlane	28	1	9	5	8	3	2	1	0
Huginn	27	5	6	1	4	8	4	1	2
Average	**28.4**	**5.6**	**3.3**	**3.3**	**4.7**	**6.1**	**3.4**	**1.5**	**2.8**

order. In this Section, we formally define two types of variants, namely simple and complex based on their nature of exhibiting the strict partial order.

Definition 5. *We refer to a set of variants as **Simple** if for any pair of variants* (ν_1, ν_2) *in the set, one variant scores not less than the other* $(\forall_i\ score_{p_i}(\nu_1) \geq score_{p_i}(\nu_2)$ *or* $\forall_i\ score_{p_i}(\nu_2) \geq score_{p_i}(\nu_1))$ *for all desired properties in a specific code context. Also, recall that* $\exists_i\ score_{p_i}(\nu_1) \neq score_{p_i}(\nu_2)$ *if* ν_1 *and* ν_2 *are variants.*

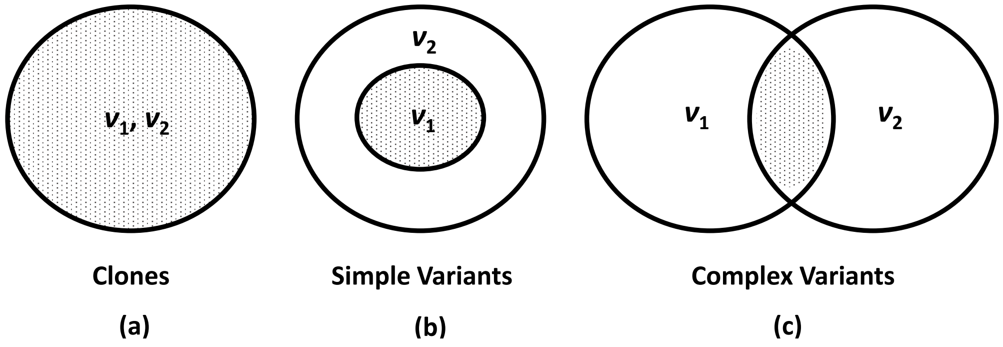


Figure 3.2: Differences between clones, simple variants and complex variants in terms of desired properties.

If ν_1 and ν_2 are simple variants in the given code context, and ν_2 is stronger than ν_1, we mean that ν_2 scores over ν_1 on at least one desired property and is as strong as ν_1 on all the other desired properties (Figure 3.2 (b)). In practice, we may find that most efficient solutions may suffer from issues such as, readability and licensing, and hence may not be better on all desired properties. As an example, consider the Internet traffic monitoring APIs, Fiddler and Titanium. A discussion on SO[3] suggests that Titanium is preferred over Fiddler given the licensing constraints. Hence code snippet containing the Titanium API is a stronger variant than the one with Fiddler API. In this case, Titanium based code is also a simple variant of Fiddler based code since it is easier to choose the prior over the latter. As another example for simple variant, an $O(nlogn)$ solution is accepted to be better than $O(n^2)$ solution where worst-case time complexity is the desired property.

Strict Partial Order The relation over the strength of code variants (represented by the symbol '>') is a strict partial order over the set $\mathcal{V}$ of variants. In other words, $\nu_1 > \nu_1$ cannot hold (irreflexivity) since we need at least one property by which the snippet being compared with should differ to be called as a variant. $\nu_2 > \nu_1$ indicates that ν_2 is a stronger variant of ν_1, and $\nu_1 > \nu_2$ cannot hold (antisymmetry), and $\nu_2 \neq \nu_1$ (irreflexivity). In addition, if we have ν_3 such that $\nu_3 > \nu_2$, then $\nu_3 > \nu_1$ (transitivity).

In the case of complex variants, it might be possible for developers to apply a weight function to choose a specific complex variant as a strong variant. Without weights or additional such preference information, it will be unclear to developers which variant to select (Figure 3.2 (c)). Figure 3.2 (a) shows the case where ν_1 and ν_2 have the same properties, no more or no less and thus they become clones.

Definition 6. *We call ν_1 and ν_2 as **Complex** variants if ν_1 scores over ν_2 for some desired properties, and ν_2 scores over ν_1 for some other desired properties. More formally, $\exists_{i,j}$ $((score_{p_i}(\nu_1) > score_{p_i}(\nu_2)) \ \wedge \ (score_{p_j}(\nu_1) < score_{p_j}(\nu_2)) \wedge (i \neq j))$.*

An example for complex variant, developers of Huginn project are seen debating over using InnoDB or MyISAM (Table 3.7). Thus, the corresponding code snippets in this context become complex variants.

[3]https://stackoverflow.com/questions/30995808

Table 3.9: Summary of discussion counts per variant type. Total refers to the total number of discussions. $\%_v$ is the Total as percentage out of 1500 discussions. $\%_d$ is the percentage out of 426 variants.

Type	*Alg*	*RO*	D_p	D_m	*Total*	$\%_v$	$\%_d$
Simple	84	49	50	70	246	16.4%	57.7%
Complex	91	51	23	42	180	12.0%	42.3%
Total	**175**	**100**	**73**	**112**	**426**	**28.4%**	

How are variants distributed across the different types of variants?

In our (discussions) dataset, developers deliberated on which variants were better suited to the project. We found that discussions about simple variants were uncomplicated, and typically involved cases where a developer proposed a variant in a patch, which was approved. Table 3.7 shows an example from the GTK+ project, where the discussion is about the computation speed. The discussion was relatively straightforward as one of the proposed algorithm was seen as faster and was supported by one developer (S2). Such discussions on simple variants accounted for 57.7% (246 out of 426: Table 3.9) of the total variant discussions[4].

On the other hand, developers deliberated on the pros and cons of the complex variants with respect to the desired properties. Table 3.7 shows an example from the Huginn project, where developers discussed the trade-off on the event processing functionality compared to the reliability of support provided. Such discussions about complex variants occurred in **42.3%** (180 out of 426: Table 3.9) of the cases of the total variant discussions.

We next analyze the kinds of discussions that occur when developers discuss the different types of desired properties. When considering Algorithmic variants, we see that there were on average 5.5% (84 out of 1500: Table 3.9) simple and 6.1% (91 out of 1500: Table 3.9) complex Algorithmic variants. We find that Apache Math had the highest number of complex RO (row 1, Table 3.8), and Fastlane had the most simple RO variant discussions (row 14). RO variants comprised of 3.2% (49 out of 1500: Table 3.9) simple and 3.4% (51 out of 1500: Table 3.9) complex variants. In SCons, we found no simple RO variant. In three of the projects (MySQL, Bootstrap and Apache Math), we found only one simple RO variant. On average, 12.3% (185 out of 1500: $D_p + D_m$ in Table 3.9) of discussions were about diction variants where 8% (120 out of 1500: Table 3.9) were simple and 4.3% (65 out of 1500: Table 3.9) were complex.

Observation

Developers seek and compare variants in 28.4% of their discussions. In 42.3% of the variant discussions, they discussed trade-offs among complex variants.

[4]Note that the reader should exercise caution while reading these numbers. A simple variant discussed in a discussion may be associated with more than one desired properties. Hence, summing up the rows in Table 3.8 is incorrect way of counting the total number of discussions annotated as about variants.

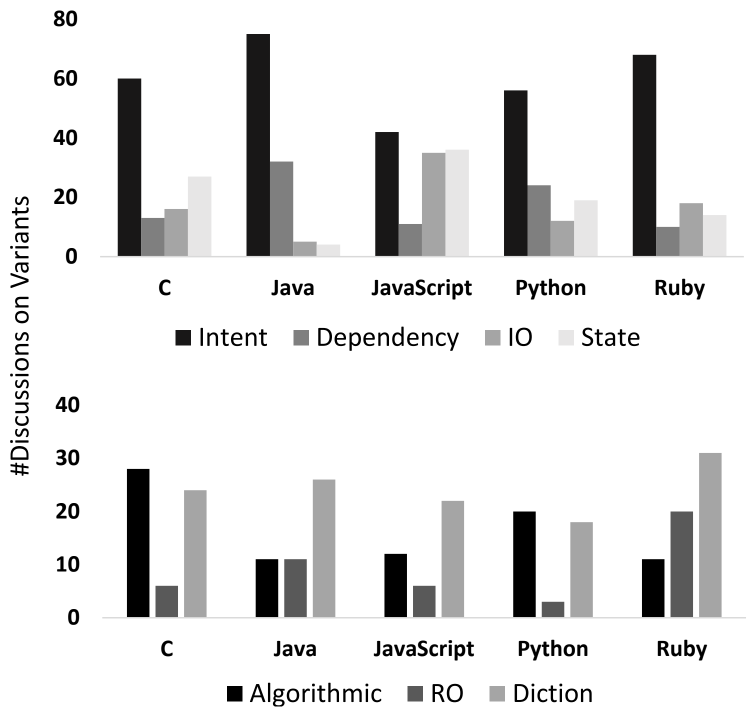


Figure 3.3: Distribution of variants across context types and desired properties over multiple languages. Diction variants are prominent in projects of all programming languages.

3.6.2 Are variants language dependent?

We next analyze to see if the constructs of a programming language impact implementation choices, and thereby the volume of variants (see Table 3.3 and 3.8).

Recall, our dataset has five programming languages, each of a different type such as object-oriented (Java), scripting (JavaScript), and procedural (C). Python and Ruby are popular general-purpose multi-paradigm languages.

In our dataset, we found examples where we find that the programming language played a factor in why developers sought variants. A Rails developer while discussing a defect[5] says, "*... might consider reverting this commit as lambdas [a construct in Ruby to execute a method whose definition is passed as string] are used a lot for lazy evaluation [which is not a desired property]*". Another JQuery developer[6] is concerned about the differences between the language API, `Deferred` and `Promise`. However, when considering the entire dataset, we do not find any statistical significance that programming language impacts the total number of variants (Kruskal Wallis, p-value=0.3782). So, next we investigate whether language plays a role in variants' desired property or code context.

Figure 3.3 shows the distribution of variants across context types and desired properties grouped by language. In Figure 3.3 we see that Diction variants occur relatively more frequently across all languages. Diction variants account for 12.3% (185 out of 1500: Table 3.9) of all discussions and 43.4% of variant discussions. However, we did not find any

[5]https://github.com/rails/rails/issues/9805
[6]https://github.com/jquery/jquery/issues/3596

statistical significance (Kruskal Wallis, p-value=0.3404) to show that Diction variants are the most affected because of language constructs. In fact, none of the desired property types correlated with language significantly (Kruskal Wallis: Algorithmic:p-value=0.3007, RO:p-value=0.4662).

Also, we do not find a statistically significant relationship between context types and languages of implementation (Kruskal Wallis: Intent: p-value=0.3601, Dependency:p-value=0.2081, IO:p-value=0.6248, State:p-value=0.5448).

Observation

We do not find significant difference between languages when comparing the volume of variants distributed across desired properties and code context.

3.7 Implications of Variant Characterization

Our characterization of variants creates opportunities for tool builders as well as researchers.

3.7.1 Tool Builders

- Developers discuss code variants in abundance, but there is a lack of relevant tools to mine variants. Automated support that can fetch re-usable code snippets will be of interest to developers.

- The large number of discussions about Diction variants suggest that there is often debate about stylistic aspects of code quality. Code refactoring tools may use this as feedback to include new stylistic or mechanical features.

- Developers need to consider the code context of variants when evaluating implementation choices. Therefore, they would benefit from automated support that can fetch relevant variants when given a code context type.

3.7.2 Researchers

- We need metrics to enable quantitative comparison of two variants. Our definitions 5 and 6 in Section 3.6.1 are based on assigning score to variants. However, deciding on the right metrics to assign such a score to variants is an open problem. Such quantification of the strength of variants is important to building variant mining tools.

- Current clone detectors are incapable of identifying variants as they do not consider the desired properties of the code. Our characterization suggests that terms that are associated with certain properties (e.g., “optimize”, “efficiency”, “accuracy” were associated with algorithmic variants) can be useful to classify discussions and in turn mine variants.

- Further research to understand the “Intent” context deeper, will help to better understand variants, as large number of variants belongs to this context type.

- Our results indicate that variants may not be dependent on language. Further research is needed to see the feasibility and potential of a language agnostic variant mining tool.
- In the domain of education, an understanding of variants could help in building feedback tools for programming assignments. In a Massive Open Online Courses (MOOC) setting, instructor can give feedback on select variants of code snippets.
- Extracting current code context for a given project has been an important step for tools that work on source code such as plagiarism detection [31] and example recommendation [24] tools. However, they focus and capture only one among the four context types. Our observations indicate that a variant mining tool will need to focus on all types of contexts for high recall.

3.8 Threats to Validity

Internal Validity We have used 15 open source projects for evaluation. We strive to control variability by selecting 100 random defects each from multiple projects. Other projects and domains might give different numbers. We have selected each project from a different domain to mitigate this threat. Also, these projects may not be representative of the entire set of open source projects. Choosing multiple projects was an attempt to reduce this kind of bias. However, we have controlled the bias due to project size and popularity. We have compared variants with their nearest type of code snippets such as clones, idioms, simions and examples. There may be more types of code snippets such as mutants. Mutants are artificially changed code used to assess the quality of test cases. They do not fit into the given code context. Hence, mutants are also not code variants.

External Validity Our results may not generalize to all types of code (for instance, scripting or functional). For evaluation, we have taken a mix of Java, C and Python projects popularly used in clone research. Hence, our work applies to high-level imperative languages at the least. Our results for variant mining depends on the discussion forum data. These results may not generalize to variants that are domain specific implementations with inadequate developer discussions.

3.9 Summary

Code variants are fundamentally different from other structural and semantically similar code snippets, such as simions and code clones. They appear frequently in source code and developer discussions. Therefore, understanding code variants is important for software development and maintenance. Currently, there are inconsistent definitions of code snippets that are similar. To the best of our knowledge, we present the first study to characterize and distinguish code variants from other types of code that are similar to each other. In this work, we define code variants, classify them as simple and complex, and categorize them into three main types based on their properties: algorithmic, diction and resource-oriented.

Our variant characterization presents several opportunities and challenges for tool support and automation.

With the availability of “big code” from open source projects and discussion forums, developers are increasingly interested in leveraging the existing implementation choices that match specific desired properties. Hence, an understanding of code variants and their characteristics can help build tools that provide automated search, including functionality such as, comparing or ranking variants, quantify the strength of variants, and building recommender systems to improve the existing development environments. In future work, we plan to address some of the open research challenges (Section 3.7) to help in building variant mining tools.

Chapter 4

Towards Searching for Code Variants

Developers often look for better choices to implement existing code. They do a web search or ask for implementation choices in discussion forums. Developers will benefit from a tool which can describe the behavior of the code snippet, and suggest alternative implementations. Towards this purpose, we build and use a knowledge base of familiar topics and their implementation choices from discussion forums to show alternative implementations for a given code snippet. We have implemented this approach in a tool called jSense. jSense constructs a knowledge base of implementation choices from SO. Using this knowledge base, jSense matching engine is able to identify the given input program and suggest alternative implementation choices with a precision of 92% and recall of 71%. Apart from the jSense tool, our major contribution includes a multi-component retrieval model and a structural representation of source code that can be used in several code search tools. This work opens up new possibilities for attacking a wide variety of challenging problems such as semantic clone detection and defect localization by exploiting structural information in source code.

4.1 Introduction

The rich programming language features and the existence of several third party libraries allow programmers to code the same requirement in multiple ways [83]. Sometimes, developers are not aware of these implementation choices [84]. Hence, they end up coding either inefficiently or even incorrectly. Implementation choices for several programming topics are discussed [85] in Question & Answer (Q & A) sites such as SO. Table 4.1 shows different implementation choices for a factorial program. The first option uses recursion. However, it uses `int` as the data type which can only hold up to 16!. The second example is designed for larger input values. The third example does simple lookups on pre-computed values assuming that the input will not be greater than 20. This implementation is used in Apache Commons [86] library. Looking at these choices, developers can make better development decisions. In SO, developers discuss these snippets and their relative merits. Developers will benefit from a tool which can understand the existing snippet and mine implementation choices for the same from such Q & A sites.

We model the problem of mining for implementation choices as a search problem over SO where input is a code snippet and the output is a set of code snippets that are "dis-

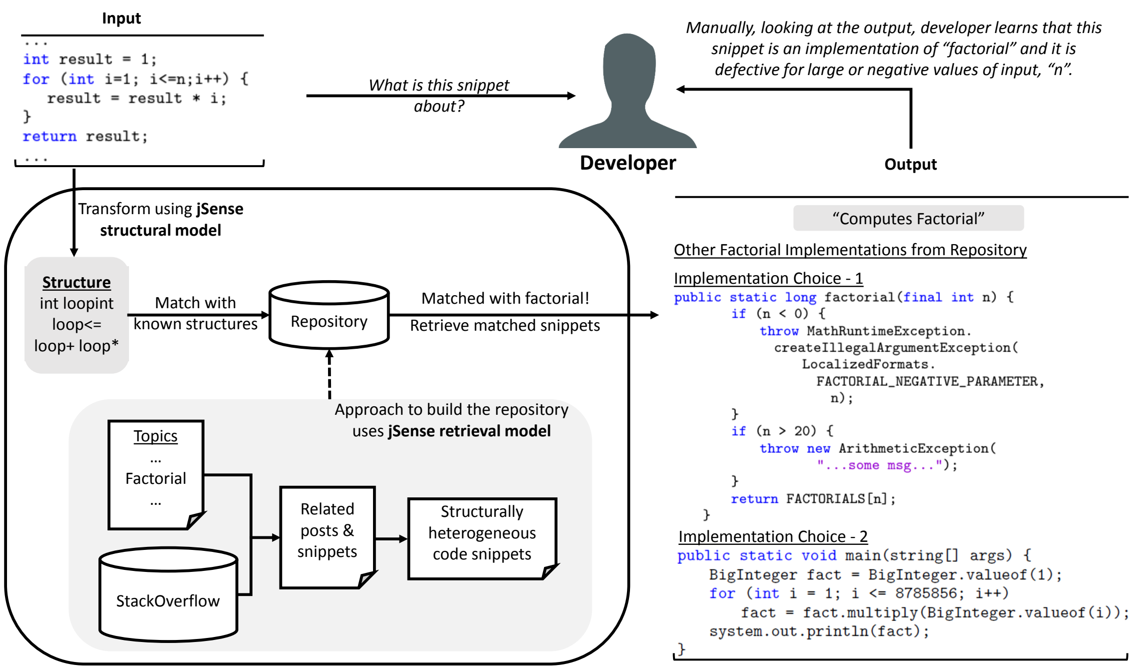
Table 4.1: Some examples of factorial implementations.

Snippet	**Remark**
`public int factorial(int n) { if (n == 0) return 1; else return (n*factorial(n-1)); }`	Commonly used text book implementation of factorial using recursion. Fails for n >= 17.
`public static BigInteger factorial(BigInteger n) { BigInteger result = BigInteger.ONE; while (!n.equals(BigInteger.ZERO)) { result = result.multiply(n); n = n.subtract(BigInteger.ONE); } return result; }`	An iterative version of factorial that uses BigInteger. This does not fail for large input values.
`public static long factorial(final int n) { if (n < 0) { throw ...Exception(..., n); } if (n > 20) { throw new ArithmeticException("...some msg..."); } return FACTORIALS[n]; }`	Factorial implementation taken from Apache Commons library avoids recomputation by doing array look up.

tinct" implementation choices. First, we propose a novel retrieval model to extract relevant code snippets from SO. Then, we de-duplicate code snippets to arrive at the distinct implementation choices. To understand which implementation choices are distinct, we need to understand the developer's perspective of heterogeneity. Towards this purpose, we conduct a user study, propose a structural model of source code (which we call jSense structural representation) and a pairwise similarity metric over this structural model.

Web search engines such as Google[1] are extensively used by developers [8, 87]. Since no special processing is done on the search engine side, user needs to query skillfully and there must exist web pages with phrases like "different ways of implementing" or "how to code without using". In a situation where the developer does not suspect his existing snippet to be buggy, he might not query at the first place. Prompter [24] shows the usefulness of SO data in such a context. They retrieve SO posts and not code snippets. Code search engines

[1]www.google.com


Input
What is this snippet about?
Developer
Manually, looking at the output, developer learns that this snippet is an implementation of "factorial" and it is defective for large or negative values of input, "n".
Output
Transform using jSense structural model
Structure
int loopint
loop<=
loop+ loop*
Match with known structures
Repository
Matched with factorial!
Retrieve matched snippets
Approach to build the repository uses jSense retrieval model
Topics
...
Factorial
...
StackOverflow
Related posts & snippets
Structurally heterogeneous code snippets
"Computes Factorial"
Other Factorial Implementations from Repository
Implementation Choice - 1
Implementation Choice - 2

```
...
int result = 1;
for (int i=1; i<=n;i++) {
   result = result * i;
}
return result;
...
```

```
public static long factorial(final int n) {
        if (n < 0) {
            throw MathRuntimeException.
              createIllegalArgumentException(
                  LocalizedFormats.
                    FACTORIAL_NEGATIVE_PARAMETER,
                      n);
        }
        if (n > 20) {
            throw new ArithmeticException(
                    "...some msg...");
        }
        return FACTORIALS[n];
    }
```

```
public static void main(string[] args) {
    BigInteger fact = BigInteger.valueof(1);
    for (int i = 1; i <= 8785856; i++)
        fact = fact.multiply(BigInteger.valueof(i));
    system.out.println(fact);
}
```

Figure 4.1: Overview of jSense design. This is a two step process: 1) We build a repository of structurally heterogeneous implementations, and 2) Use this repository for suggesting implementation choices for the identified topic.

such as CodeExchange [88] are useful in getting code snippets directly instead of web pages. However, to get distinct choices, a developer needs to read several snippets. jSense can suggest different ways of implementation for the existing code context. Thus, our work is different from these existing works.

As an example of our approach, we implemented a system called jSense. jSense indexes the posts tagged as Java in SO. It constructs a knowledge base (KB) of known topics and implementation choices using a novel Multi-Component Multi-Aspect Term Frequency retrieval model. While indexing, jSense uses the structural representation to eliminate duplicate implementation choices. To improve the efficiency of matching the input code structure with the structures in KB, jSense uses MinHash [89]. If a match is found, it returns the topic which is a phrase to identify the behavior of this snippet in natural language (NL) terms associated with the structure. It also shows all the other heterogeneous snippets from the KB associated with that topic. Figure 4.1 gives an overview of jSense design using a shorter and more familiar example of "factorial".

Our Contributions Major contributions of this work are:

- **Approach:** A search based approach to identify the given code snippet, and show its alternative implementations.
- **Retrieval and Structural Models**:
  - We propose a novel retrieval model called the Multi Component Multi-Aspect Term Frequency Model to build a knowledge base of code snippet structures.

  - We propose an improved structural representation of source code structure to enable us compute relevance of given snippet with the knowledge base.

- **Tool**: We implement and evaluate our findings using a tool named jSense.

4.2 jSense

We perceive an SO post as a combination of three heterogeneous components namely, title, body and text. Title and body are NL components whereas code has different structure and semantics. Therefore, we aim to adapt the MATF model to suit this context.

4.2.1 jSense Retrieval Model - MC-MATF

Here, we introduce a multi-component formulation of MATF which we refer to as MC-MATF. SO has three key components: title, body and code in each post. To generalize, let us assume that there are n components. We compute the MATF score for each component. MC-MATF is a geometric mean of the individual weighted MATF scores thus computed for each component. In the scenario that one component does not have any of the query terms, we do not want the overall MC-MATF score to be zero. To avoid this scenario, we introduce a smoothing parameter α. Equation 4.1 gives this formulation. We have used the ideas of add-one or Laplace smoothing but with a much smaller value of $\alpha = 0.01$.

$$tf_{MC}(d, q) = \left(\prod_{c=1}^{n} [\gamma_c tf_{M_c}(t, p_c) + \alpha] \right)^{\frac{1}{n}} \tag{4.1}$$

where MATF score $tf_{M_c}(t, p_c)$ for each component is given as:

$$tf_{M_c}(t, p_c) = w_c \times \frac{tf_{R_c}(t, p_c)}{1 + tf_{R_c}(t, p_c)} + (1 - w_c) \times \frac{tf_{L_c}(t, p_c)}{1 + tf_{L_c}(t, p_c)} \tag{4.2}$$

MC-MATF score for a document (tf_{MC}) The MATF score for each component is computed using the tf_M formulation in Equation 4.2. The parameter p_c denotes a specific component in the post such as title. We use the notation tf_{R_c}, tf_{L_c} and w_c to denote component-wise scores for RITF, LRTF and aspect weight respectively. We use the same $\frac{2}{1+log_2(1+|Q|)}$ formulation for the aspect weight w_c as discussed in Section 2.1.2. The MC-MATF score is then a geometric mean over the weighted MATF scores of each component.

Component Weight (γ_c) Different components carry different amount of term saliency in the corpus. For instance, in the case of SO posts, developers ensure that the same question is not asked before by searching through several titles of existing posts. Therefore, they carefully choose relevant terms for the title. Hence, title terms should carry a higher weight compared to the rest of the components in SO. We use γ_c to refer to the weight of each component.

Term Frequency (tf_{L_c}, tf_{R_c}) The LRTF (tf_{L_c}) and RITF (tf_{R_c}) values correspond to the length normalized and simple counts of terms in the document, respectively. This works for text. However, for source code, we need to do additional processing to extract the tokens of interest. For source code, we extract the AST and from there, we get the identifier tokens. Each identifier is stemmed and processed for stopword removal. The resulting tokens of code constitute the terms of the component.

We apply the standard IDF measure. We compute the similarity between query and post vectors, as follows:

$$SIM(Q,p) = \sum_{i=1}^{|Q|} tf_{MC}(q_i,p) \times \log_2\left(\frac{N}{df(q_i)}\right) \tag{4.3}$$

A reasonable TF-IDF model should satisfy Fang's constraints [90]. Hence, we validate MC-MATF against these constraints. TF constraint states that "*Let* $q = w$ *be a query with only one term* w. *Assume there are two documents of same length,* $|d_1| = |d_2|$. *If* $count(w,d_1) > count(w,d_2)$, *then* $tf(d_1,q) > tf(d_2,q)$". The function, $count(w,d_1)$ refers to the count of words w in document d_i. While proposing this constraint, they assume that the documents are homogeneous. We extend this constraint to multi-component corpora as follows:

- *Multi-Component TF Constraint 1*: Assume $|d_1| = |d_2|$. If $\forall_{c=1}^{n}$ $(count(w,c,d_1) > count(w,c,d_2))$, then the term frequencies $tf(d_1,q) > tf(d_2,q)$. Here, $count(w,c,d_j)$ refers to the count of word w in a component c of document d_j.

MC-MATF satisfies this modified constraint. Similarly, the following constraint holds for MC-MATF:

- *Multi-Component TF Constraint 2*: This constraint ensures that *tf* increase is lesser for larger TF values when the word count increase remains the same. For example, a word count increase from 1 to 2 contributes more to *tf* than an increase from 100 to 101. Let $q = w$ be a query with a single term w. Assume that $|d_1| = |d_2| = |d_3|$ and $\forall_{c=1}^{n}$ $count(w,c,d_1) > 0$. If $\forall_{c=1}^{n}(count(w,c,d_2) - count(w,c,d_1) = 1)$ and similarly $\forall_{c=1}^{n}(count(w,c,d_3) - count(w,c,d_2) = 1)$, then we should find that $tf(d_2,q) - tf(d_1,q) > tf(d_3,q) - tf(d_2,q)$.

Fang et al. [90], also propose a length normalization constraint and a IDF constraint. Since we use the standard IDF measure and MATF length normalization, those constraints do not change.

4.2.2 Structural Model

Our aim is to index and query structural information in source code. Source code comprises of structure expressed by the syntactic tokens. Research literature shows several ways of representing the structure ([91, 92, 93, 31]). Some of them are very selective in the information captured, such as Control Flow Graphs (CFG) [94]; while some are too descriptive such as Abstract Syntax Trees (AST) [94].

Selective Set Method In order to find a sweet spot between achieving scalability (to index millions of code snippets) and precision (in retrieval), a bag of tokens representation of source code is suitable for retrieval which selects structural tokens based on the developer's perspective of heterogeneity.

In this work, we define any subsequence of tokens of a compilable program as a *Code Snippet*. Therefore, generally code snippets are not error-free compilable units. For example, method definitions or just a sequence of statements are typical code snippets shared on SO. The tokens can be structural (ζ_{ss}) or user-defined (ζ_{voc}). Examples of ζ_{ss} are tokens corresponding to loops and branches. Tokens in the variable names, method names and comments are examples of ζ_{voc}. A code snippet $s = (e_1, e_2, ..., e_n)$ is a sequence of n tokens where $e_i \in \zeta_{ss} \cup \zeta_{voc}$.

A structural representation $\tau \in T$ for a code snippet $s \in S$ is obtained using a function $R : S \mapsto T$ which maps a snippet to a desugared searchable sequence of abstracted structural text. In other words, R is an AST reduction which converts $s = (e_1, e_2, ..., e_n)$ to $\tau = (t_1, t_2, ..., t_m)$ where $m <= n$ and $t_i \in \zeta_{st}$. The set of abstracted structural tokens ζ_{st} is defined as follows:

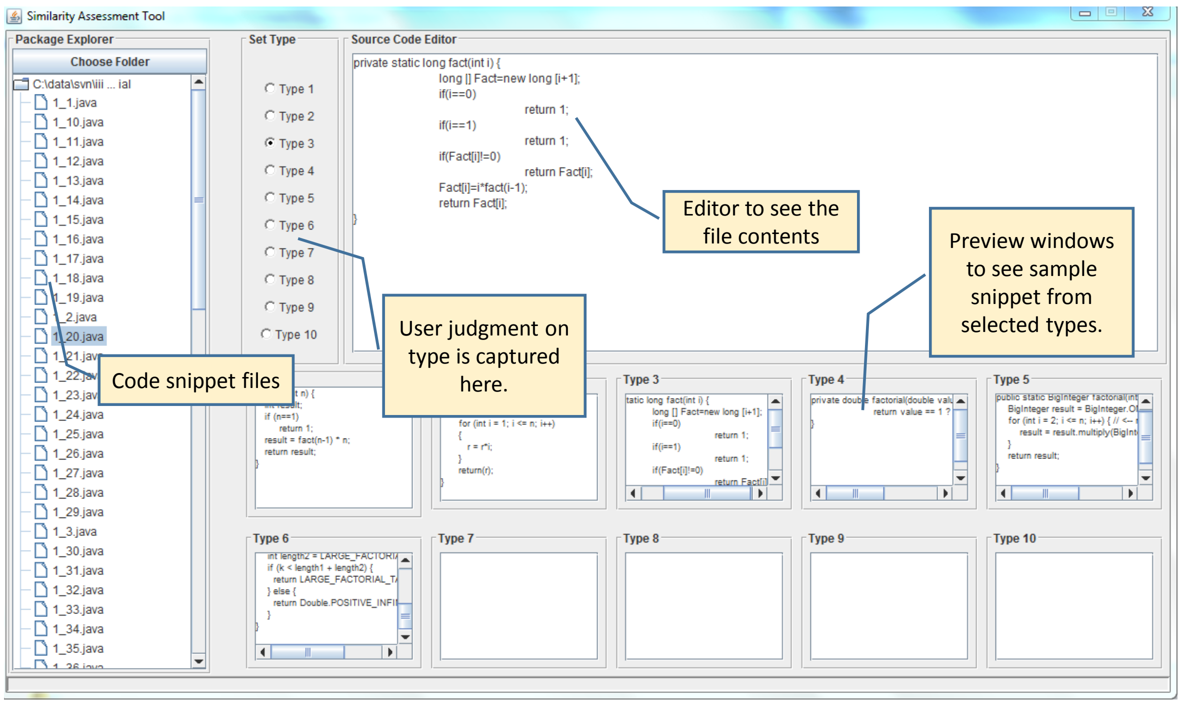
$$\zeta_{st} = \{\text{``}\textit{loop}\text{''}, \text{``}\textit{branch}\text{''}, \textit{operator}, \textit{nested}\} \tag{4.4}$$

Here, *"loop"* refers to looping structures such as `for`, *"branch"* refers to conditionals such as `if` in Java, *operator* refer to the canonicalized operators found in the snippet. Java allows operators such as "`x *= y`" that can be canonicalized to "`x = x * y`". Nested structural tokens such as a loop (such as `for`) inside a conditional (such as `if`) are represented by concatenating abstracted structural tokens governed by the regular expression $(\text{``}\textit{loop}\text{''} | \text{``}\textit{branch}\text{''})^{+}\textit{operator}$ wherever there are loops or branches. We refer to them as *nested* in Equation 4.4. We navigate the AST of the given source code and use this set of elements to synthesize a textual abstraction. Thus, we arrive at a structural representation string phrase of the input factorial code shown in Figure 4.1 as $\tau =$ `(loop<= loop+ loop*)`. We improve this model in Section 4.2.3 towards building a better repository.

4.2.3 Refining the Structural Model

To understand the code similarity, we conducted a study with thirty graduate teaching assistants from IIIT Delhi. Henceforth, we consider them as proxy to software developers. We ensured that they have at least two years of Java coding experience. They were given 77 snippets of "Factorial", 62 snippets of "Palindrome", and 70 snippets of "Reversing a String". The choice of the topics were driven by mainly two criteria: a) All the variant implementations of this topic should be easy to understand for the developers, and b) The variant implementations should exist in large numbers in tutorial sites, real-world projects and discussion forums. We wanted the focus of developers to be on the structure and not on the algorithm or API usages. So, we looked at a popular tutorial site[2] named javaTpoint and picked three topics from their top-5 programming examples. We confirmed that the code is used in large real-world projects (such as Apache Commons Mathematics Library [95]) and in discussion forums (such as SO). We extracted top-30 results from CodeExchange (which

[2]http://www.javatpoint.com/java-programs,
http://www.javatpoint.com/factorial-program-in-java



```
private static long fact(int i) {
                long [] Fact=new long [i+1];
                if(i==0)
                                return 1;
                if(i==1)
                                return 1;
                if(Fact[i]!=0)
                                return Fact[i];
                Fact[i]=i*fact(i-1);
                return Fact[i];
}
```

```
t n) {
    if (n==1)
        return 1;
    result = fact(n-1) * n;
    return result;
}
```

```
    for (int i = 1; i <= n; i++)
    {
        r = r*i;
    }
    return(r);
}
```

```
tatic long fact(int i) {
        long [] Fact=new long [i+1];
        if(i==0)
                        return 1;
        if(i==1)
                        return 1;
        if(Fact[i]!=0)
                        return Fact[i]
```

```
private double factorial(double valu
                return value == 1 ?
}
```

```
public static BigInteger factorial(int
    BigInteger result = BigInteger.O
    for (int i = 2; i <= n; i++) { // <--
        result = result.multiply(BigInt
    }
    return result;
}
```

```
    int length2 = LARGE_FACTORI
    if (k < length1 + length2) {
      return LARGE_FACTORIAL_T
    } else {
      return Double.POSITIVE_INFI
    }
}
```

Figure 4.2: A snapshot of the tool we used in our study of structural similarity is shown here. Users categorized the given snippet into 1 to 10 types based on their judgment.

is a search tool that indexes GitHub projects), next 30 by doing a Google search for these implementations in projects of at least 1000 lines of code, next from top-15 posts of SO. Since these topics were picked up from a tutorial site, we also used their implementations. Note that javaTpoint shows multiple implementation choices for implementing the same topic. There was no preference for a specific implementation choice, project, or a specific developer.

The developers were shown all the code snippets collected for one topic. They were asked to pick heterogeneous examples. Figure 4.2 shows the tool which they used to classify the snippets. They were asked specifically to classify snippets into 3 to 10 types. The code snippets used in the study along with the developer annotations are shared on our website[3].

Thirty developers annotated the data. This gave us 6270 (=(77+70+62)*30) data points to study. Using these judgments we calculated the pairwise similarity of code snippets as the normalized number of users who put the snippets (s_1 and s_2) in the same type as follows:

$$Similarity(s_1, s_2) = \frac{|users(s_1, s_2)|}{|users|} \tag{4.5}$$

4.2.3.1 Reduction Rules

Recall that the purpose of the study was to understand the developers' perspective of structural similarity. We seek to discover the set of rules which can reduce two code snippets to the same structural representation if they are judged to be belonging to same group by the

[3]http://jsense.epizy.com

developers. From the study, we observed that the following reduction rules are useful for structural comparison of code snippets.

Rule 1: Identifier and Literal Removal We observe that renaming identifiers and literals do not cause developers to consider the snippets as different. Hence, we drop them from the structural representation. Table 4.2 (a) shows an example where dropping `n`, `num` and `fact` makes the snippets identical .

Rule 2: Idiomatic Replacements There are language supported syntactic sugars that allow different idioms to replace each other without affecting the structure of the code. For example, replacing `while` loop with a `for` loop does not affect developer's score of structural similarity (see Table 4.2 (b)).

Loop and Branches are two common idioms found in Java. Loops appear in three different forms (`for`, `while` and `do`). Branches appear in three forms (if-else, switch-case and ?:).

Top-k structural elements A BigInteger based factorial implementation must be distinguished from an Integer based implementation and hence we see that *type* information plays a significant role in structural representation of a code snippet. *Method calls* are typically informative and hence we choose to include it in the set of elements we consider. Use of different libraries provide different interesting applications which could be alternative implementations. *Recursion* is a popular technique for repeated execution and backtracking. Therefore, we modify Equation 4.4 as follows:

$$\zeta_{st} = \{\text{“}loop\text{”}, \text{“}branch\text{”}, operator, type, methodCall, \text{“}recursion\text{”}, nested\} \tag{4.6}$$

We tokenized the snippets and used the major structural elements that distinguish heterogeneous snippets from the others. The top-6 features turned out to be: loops, branches, operators, types, method calls and recursion (Equation 4.6). Loops, branches and operators are already covered in Equation 4.4. With this improved model, the definition of *nested* in terms of regular expression becomes:

$$nested = (\text{“}loop\text{”}|\text{“}branch\text{”})^{+}(operator|type|methodCall|\text{“}recursion\text{”}) \tag{4.7}$$

Rule 3: Merging Declarations For readability reasons, developers tend to keep one declaration per line. Having multiple declarations in the same line, or using consecutive lines, is just a matter of style. It does not affect the code structure. Hence, we propose a reduction rule to canonicalize all variable declarations to one per line. Moreover, the order of their occurrence does not matter either, as long as the declarations belong to the same scope. So, we sort them alphabetically as part of this reduction. See Table 4.2 (c) for an example of this reduction.

Rule 4: Stop Word Removal Removal of some tokens improves the similarity score between snippet pairs. In Table 4.2 (d), the line `System.out.println(...)` does not affect the structure, as per our study. Hence, their removal helps similarity calculation.

Table 4.2: We have used four reduction rules. Here, we show two code snippets that reduce to an identical snippet after the application of reduction rule.

Before:

```
if (num<=1)
  return 1;
else
  return num *
      fact(num - 1);
```

Before:

```
if (n <= 1)
  return 1;
return
  n * f(n - 1);
```

After:

```
if ( <= )
   return ;
return * ( - );
```

(a) Removal of identifiers and literals.

Before:

```
int =
for (...)
  { *=
  }
return
```

Before:

```
int =
while (...) {
  *= }
return
```

After:

```
int =
loop (...) { *= }
return
```

(b) Idiomatic Replacement.

Before:

```
int n = 5;
int fact = 1;
for (int i=1;
      i<=n;i++)
  fact = fact * i;
```

Before:

```
int n, fact = 1;
n = 5;
for (int i=1;i<=n;i++)
  fact = fact * i;
```

After:

```
int = =
for (int = <= ++)
   = * ;
```

(c) Merging and Sorting Declarations.

Before:

```
int fact = 1;
for (int i=num;i>1;i--){
  fact = fact*i;
  }
System.out.println(fact);
```

Before:

```
int fact = 1;
for (int i = num; i>1; i--){
  fact = fact*i;
  System.out.println(fact); }
```

After:

```
int fact = 1;
for (int i = num; i>1; i--){
   fact = fact*i; }
```

(d) Removal of lines containing println.

Table 4.3: Some examples of jSense structural representation after the application of reductions and structure flattening.

Snippet	jSense Representation
`public int factorial() { if (n == 0) return 1; else return (n*factorial(n-1));}`	`int branch== branchfactorial branch− branchrecursion branch∗`
`public int factorial() { return n == 0 ? 1 : n * factorial(n - 1); }`	`int branch== branchfactorial branch− branchrecursion branch∗`
`public int factorial() { int result = 1; for (int i = 2; i <= n; i++) { result *= i; } return result; }`	`int int loopint loop<= loop+ loop*`

There are several such stop-word tokens such as `return`, delimiters (`;`), log statements, and comments. Hence, instead of identifying all these tokens, we follow a white-listing approach to these top-5 tokens for the structural representation to get better results. Table 4.2 (d) gives an example.

In most high level languages, some structural elements such as loops and branches are containers defining new block for more statements. We can capture this containment by appending tokens within the block with the container token. For instance, a loop containing an initialization shows up as: `for (int i = 0; i<10; i++)` { `int temp = ...`}. We flatten it to `loopint loop= loop`< `loop++ loopint loop=` after applying the reduction rules.

From the study, we find that loops and branches from the reduced code are the major containers of structural elements. Hence, we restrict flattening only to these two structural elements. Table 4.3 shows some example code snippets and their corresponding jSense structural representation.

4.3 Implementation

Our implementation comprises of two steps: 1) Building the repository, and 2) Querying it for known structures. In this Section, we describe them in detail.

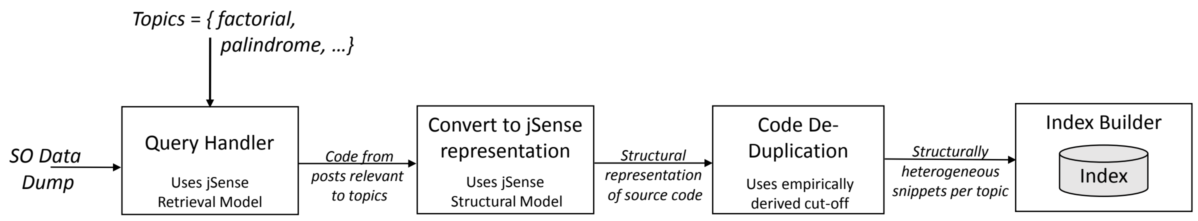

Topics = { factorial, palindrome, …}
SO Data Dump
Query Handler
Uses jSense Retrieval Model
Code from posts relevant to topics
Convert to jSense representation
Uses jSense Structural Model
Structural representation of source code
Code De-Duplication
Uses empirically derived cut-off
Structurally heterogeneous snippets per topic
Index Builder
Index

Figure 4.3: We build a repository of structurally heterogeneous implementations for a given set of topics. We use jSense retrieval model to find relevant posts from SO. We use jSense structural model for computing code similarity.

4.3.1 Building the Repository

We consider each SO post (a single answer) with its title as one document. We used the Porter Stemming [96] and a stopwords list comprising 173 terms[4] to pre-process the terms in the post. We used jSense Retrieval Model (see Section 4.2) to retrieve top-10 posts from SO relevant to each query topic. Since our jSense Retrieval Model is an MATF implementation, it can be integrated with existing IR tools such as the Lemur toolkit[5]. We assume that every code snippet in the most relevant posts are relevant implementations for the given topic. This is not always true with SO posts. However, we find it a reasonable assumption to make. The noise in the results at this stage gets eliminated to a large extent by the next two steps in the process. We convert the code snippets to their structural representation using our jSense Structural Model (see Section 4.2.3). Finally, we compute pairwise similarity to cluster code snippets. We use a cut-off (ψ) to discriminate snippets as similar or dissimilar. We retain the first result from each cluster of similar snippets, and drop the rest. Thus we get structurally heterogeneous code snippets per topic. We call this as the repository. Figure 4.3 gives an overview of this sequence of steps.

Extracting Structure from Code Snippets Extracting structure as per the reduction rules, and representing them in a searchable format is at the heart of our approach. We use EclipseJDT[6] to extract the structural elements from Java programs by navigating through its AST. All the structural elements listed in Equation 4.6 except recursion can be extracted by AST navigation.

While we navigate through the AST, we append the value of type (for instance, int, char, or Student), the name of the method being invoked, and the term "recursion", as we encounter each of them. We separate them by space. If we encounter an operator, we check if it is inside a loop or a branch. In the first case, for loop, we append "`loop`" and the operator (for instance, `loop+`), and for branch, we append "`branch`" and the operator (`branch<`). For the latter case, we just append the operator to our structure string.

To identify recursion, we search for method name within the method definition. This is not an accurate way of detecting recursion. However, we rely on the assumption that the

[4]http://www.ranks.nl/stopwords
[5]https://www.lemurproject.org/
[6]http://www.eclipse.org/jdt/

method name would rarely appear in the method definition given that we pre-process the snippet to strip comments.

For each topic, over the collected code snippets, we use the length of the jSense structural model (the number of tokens in them) to remove outliers. We can observe that, empty snippets have zero length. A large code snippet of say 2000 lines of code will have proportionally higher length in their corresponding jSense structural model. The intuition is that the variants of similar features will have similar structural representation. Hence, we can drop snippets which have jSense structural model of length two standard deviations away from their mean length.

4.3.2 Querying the Repository

Given any Java input program, we used Eclipse JDT to extract its method definitions. We pre-processed the code snippet by cleaning line and block comments. Next, we transform them to their jSense structural representation. The structural representation of the input snippet becomes the query. In the repository, for each topic, we store the corresponding structural representations and code snippets. To calculate the relevance of input structure to the structures in repository, we use Jaccard similarity (JS) [97] over the sets of tokens in these structural representations (τ) as shown below in Equation 4.8:

$$JS(s_1, s_2) = \frac{|\tau_1 \cap \tau_2|}{|\tau_1 \cup \tau_2|} \tag{4.8}$$

Note that for input code snippet pair (s_1, s_2), similarity is computed on the corresponding structural tokens (τ_1, τ_2). To preserve sequence information, we apply Equation 4.8 on tri-grams of the structural representation.

Computing Jaccard Similarity during query time is prohibitively expensive. Hence, we resort to an approximation. In our implementation, we approximate JS using Min-Hash [89] technique. The approximation error (ϵ) can be theoretically bound to $O(\frac{1}{\sqrt{k}})$ for k hash signatures. As we increase the hashes, the approximation error reduces. To bound the expected value of ϵ to a small value of $\mathbb{E}[\epsilon] < 0.05$, we use 400 hashes.

For all matching structures where the similarity is higher than a specific cut-off (same value as ψ), we extract the corresponding code snippets and topics. It is possible that same structural representation matches with snippets from different topics. When we get multiple results, we use a ranked list of vocabulary items associated with each topic for disambiguation.

The resulting topics with best match on similarity after disambiguation are shown to the users along with the corresponding snippets as alternative implementation choices.

4.4 Evaluation

In this evaluation, our goal is to understand the effectiveness of the jSense tool. In that process, we seek to understand the role played by the structural and retrieval models in the overall performance of the tool. First, we empirically derive the parameters for structural comparison and thereafter evaluate the effectiveness of our approach to build a repository.

We evaluate the retrieval models considering SO as a collection of posts with each post containing three components namely title, body and code for each post. We evaluate the overall approach by computing precision and recall for 12 topics. We compare our system with a state of the art code search tool named CodeExchange [98, 88]. Finally, we interview industry developers to understand the usefulness of our tool.

Experimental Setup There exists 2.7 Million posts tagged as Java in SO[7]. From the SO posts, we extract all the three components namely title, body and code for every post tagged as Java. We remove stop words and apply Porter's [96] stemming to the text. We ignore images and other non-text content from the SO posts that contain code snippets satisfying the above criteria.

The code snippets embedded in SO posts are partial programs. For example, developers may sometime leave ellipses ("...") and only provide lines of interest. To be able to result in code snippets that are more meaningful, we drop the code snippets that do not match the following criteria:

- *Correctness Criteria*: We could extract AST out of the snippet. We should be able to navigate the AST and extract the structural elements such as conditionals and loops from the code snippets. It is possible that the "<code></code>" tag within which code snippets usually appear contain just a stack trace or non-Java code.
- *Adequacy Criteria*: There must be at least three lines of code. The information content in the structure of code snippets must have the capacity to discriminate one code snippet from the other implementing a different topic.
- *Usefulness Criteria*: We retrieve code snippets that are either a set of lines or a single method definition. We drop snippets if they contain class definition or multiple methods. This improves precision in retrieving reusable snippets.

Applying these criteria results in 94,449 Java tagged SO posts which we use to build the repository.

4.4.1 Building the Repository

We index all the 94,449 method definitions using MC-MATF. From the posts that contain these methods, we use Mallet[8] to arrive at topics. We manually review the top 500 topics list for those topics that can potentially have code snippets implemented as a single Java method. Thus, we found 156 potentially useful query topics. Using the associated code snippets automatically retrieved using the jSense retrieval model, we build the jSense repository. We also manually annotate the code snippets in SO to form the gold set. This set of snippets act as a gold set for evaluation allowing us to compute precision and recall. We have used single method Java snippets only as a gold set preparation strategy. Our approach and tool can work with any code snippet.

[7]We use 09/2016 version of SO from https://archive.org/download/stackexchange/stackoverflow.com-Posts.7z.

[8]http://mallet.cs.umass.edu/topics.php

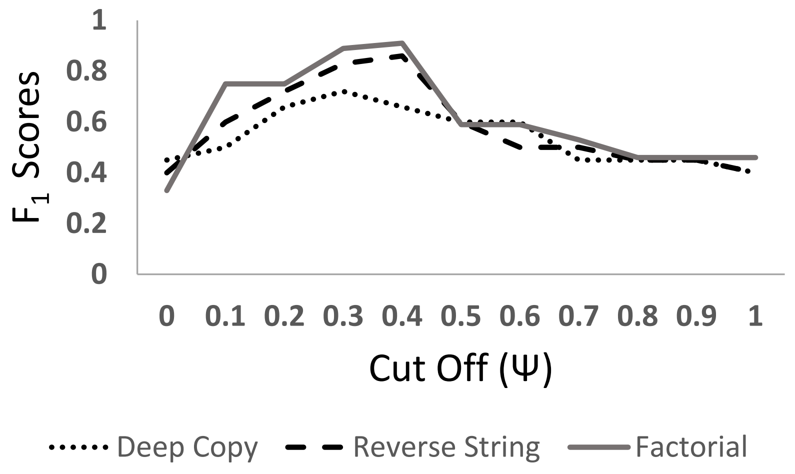


Figure 4.4: While de-duplicating the results, for different values of ψ, we note the corresponding $F_1 Score$ for each topic. We observe that a cut-off of 0.4 yields best results.

On an Intel Xeon E5 workstation with 16 GB RAM and 3.2 GHz CPU, the average time taken over 20 attempts to build the repository with 156 topics was 118 minutes. Loading the repository without any optimization took 8 minutes. Thereafter, jSense gave a sub-second response time for queries. These steps can be further optimized for both space and time. Since this is a one-time offline process, optimizing it is not our focus.

De-duplication using Structural Model A key step in the process of building the repository is to remove the structurally similar snippets from the relevant snippets retrieved from SO. We refer to this step as de-duplication. When we match the structure of input snippets for de-duplication, we use a threshold to cut-off (ψ) the results which are structurally similar. Increasing ψ brings non-heterogeneous snippets. Recall suffers if we decrease ψ. For example, on analysis of results with $\psi = 0.2$, we observe that not all types are important in classifying snippets as heterogeneous. Table 4.4 shows results for various ψ values. We thus find that $\psi = 0.4$ works best for our context. Figure 4.4 shows the relationship between F_1 Score and ψ for three topics.

We compare our structural model with the reduced set-based model [37] over the same set of topics. Three developers with at least two years of experience of working with Java language manually evaluated the resulting code snippets as relevant and heterogeneous. We have improved the precision from 83% to 92%. The inter-rator agreement was 100% in this case which we attribute to the simplicity of the topics. This improvement is because of two main reasons: a) the inclusion of more structural elements in the jSense model, and b) the improved relevance of results due to MC-MATF while building the repository.

Retrieval Models - Comparison between TF, MATF and MC-MATF The quality of snippets in our repository is a result of the effectiveness of the retrieval model. Replacing the retrieval model affects the overall precision and recall. As shown in Table 4.7, MC-MATF gives the best results. We attribute the performance to the ability to distribute

Table 4.4: A comparison of snippets retained by our approach against a developer's hand-picked set of implementations, for the *"factorial"* topic.

Cut-off (ψ)	**Snippet Count**	**Precision**	**Recall**	**F_1 Score**
0.0	1	1.00	0.20	0.33
0.1	3	1.00	0.60	0.75
0.2	3	1.00	0.60	0.75
0.3	4	1.00	0.80	0.89
0.4	6	1.00	0.83	0.91
0.5	6	0.42	1.00	0.59
0.6	6	0.42	1.00	0.59
0.7	6	0.36	1.00	0.53
0.8	6	0.29	1.00	0.46
0.9	6	0.29	1.00	0.46
1.0	6	0.29	1.00	0.46

weights across the multiple components in SO. SO posts have a wide distribution of distinct words and post length (see Section 2.1.2). Hence, MATF works better than TF in the SO context. Table 4.5 evaluates jSense with MC-MATF on 12 short queries. We replaced them with 30 long queries of length ranging from 5 to 20 with a mean and median of 14. We used $\gamma_{body} : \gamma_{code} = 2 : 1$ ratio to calculate the precision and recall. The results are still similar. MC-MATF outperforms both MATF and TF. Moreover, the results that MATF performs better than TF justify our decision to adapt MATF. Drop in F_1 scores for long queries is consistent across retrieval models. Note that these results are after the application of stemming and stop word removal on long queries.

4.4.2 Querying the Repository

Since the repository is built using SO data, we chose input code snippets from a different source. We collected Java programs used in the most popular Java books. We use this as our input dataset for evaluation. Since SO is typically used by new programming language learners, we used top three Java books as our test dataset for code snippets. We downloaded the code snippets from the website. The top three books from a leading online shopping site were *Head First Java*, *Effective Java* and *Java: A Beginner's Guide*. The rationale for test data selection has no relation to the retrieval approach.

Comparison with CodeExchange To understand the overall effectiveness of the jSense tool, we compare the tool output with the results of CodeExchange [88] (CE) which is a code search tool. CE indexes GitHub code snippets. It expects the input to be an NL query. In our case, we use a code snippet as the input. We extract the identifiers from the code snippet and use them as the query (referred as CE_b in Table 4.5 and Table 4.6) and thereafter compare the results. In some queries such as "compare objects" the identifiers in the code are not very useful for search. Therefore, to boost the results of CE_b, we query with the topic string mentioned in Table 4.5. We have reported the best results out of both

Table 4.5: We evaluate jSense by comparing the results with a goldset. *GS* represents the gold set. JS_b and CE_b represent the baseline versions of jSense tool and CodeExchange *CE* respectively.

	# Distinct Choices					**# Retrieved**				**# Relevant**			
Topic	*GS*	JS_b	JS	CE_b	CE	JS_b	JS	CE_b	CE	JS_b	JS	CE_b	CE
binary search	3	1	2	1	1	10	4	10	10	3	4	2	2
check file exists	5	2	3	2	2	10	4	10	10	8	4	6	8
compare objects	5	2	4	0	1	10	10	10	10	10	10	4	10
string to int	7	0	0	0	0	10	10	10	10	0	0	0	0
deep copy	4	2	2	1	1	10	6	10	10	9	6	9	9
factorial	7	4	5	3	4	10	10	10	10	9	10	9	9
file reading	6	2	6	1	1	10	7	10	10	8	7	6	8
keyboard input	4	2	4	1	2	10	10	10	10	10	10	7	10
merge sort	3	1	3	1	1	10	5	10	10	9	5	8	9
check palindrome	7	2	5	2	2	10	8	10	10	6	8	5	5
reverse string	7	2	5	2	2	10	10	10	10	6	10	3	3
sorting array	6	3	5	1	1	10	10	10	10	10	10	10	10

of these query formulations as *CE* in Table 4.5 which compares the precision and recall of CodeExchange with jSense. We use JS_b to denote a baseline jSense system with $\psi = 0.2$ and $\gamma = 1$ for all components. *JS* denotes a jSense system whose parameters are empirically derived. When compared to *CE*, a well-configured jSense system improves the precision from 69% to 92%. We empirically derive $\psi = 0.4$. Component weights distribution at a ratio of $\gamma_{title} : \gamma_{body} : \gamma_{code} = 4 : 2 : 1$ gives best results which reconfirms that title terms should be more important than the body of the post. Terms in code are usually shortened identifiers such as "fact" for "factorial". jSense outperforms CodeExchange in all queries except the *int to string* case in terms of F_1 Score. All retrieval models based on the bag of words assumption lose precision for the query, *int to string* which is semantically different from *string to int*. In another example, query terms such as "binary" and "search" appear independently in several posts and not necessarily in posts discussing "binary search". Hence, *CE* and JS_b score poorly on precision. *JS* gives higher weight to terms in title thereby gains precision. Clone detectors such as SourcererCC [99] and Deckard [74] are also based on token similarity. However, these tools do not allow us to optimize the query as we did for *CE*. Moreover, *CE* is a hosted tool which is already optimized for search.

User Study on Effectiveness of jSense To evaluate the usefulness of this tool, we interviewed nine industry developers with at least two years of development experience in leading software development organizations. In the first part of the study, we showed them a code snippet. We asked them the following three questions: a) to explain its behavior, b) if they would have coded the feature that way, and c) whether they find any potential bugs or missing parts. In the second part, after gathering their answers, we showed them five alternative implementations for the same snippet as generated by jSense from SO and asked them the same questions again. This time around, they identified 57% more issues. 70% of the developers said that they would have written the code differently after looking at the

Table 4.6: We extract implementations with an average precision of 0.92 compared to the 0.69 of CodeExchange (CE). jSense (JS) improves recall from 0.29 CodeExchange (CE)to 0.71 (JS).

	Precision				**Recall**			
Topic	JS_b	JS	CE_b	CE	JS_b	JS	CE_b	CE
binary search	0.30	1.00	0.20	0.20	0.33	0.67	0.33	0.33
check file exists	0.80	1.00	0.60	0.80	0.40	0.60	0.40	0.40
compare objects	1.00	1.00	0.40	1.00	0.40	0.80	0.00	0.20
string to int	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
deep copy	0.90	1.00	0.90	0.90	0.50	0.50	0.25	0.25
factorial	0.90	1.00	0.90	0.90	0.57	0.71	0.43	0.57
file reading	0.80	1.00	0.60	0.80	0.33	1.00	0.17	0.17
keyboard input	1.00	1.00	0.70	1.00	0.50	1.00	0.25	0.50
merge sort	0.90	1.00	0.80	0.90	0.33	1.00	0.33	0.33
check palindrome	0.60	1.00	0.50	0.50	0.29	0.71	0.29	0.29
reverse string	0.60	1.00	0.30	0.30	0.29	0.71	0.29	0.29
sorting array	1.00	1.00	1.00	1.00	0.50	0.83	0.17	0.17
Average	**0.73**	**0.92**	**0.58**	**0.69**	**0.37**	**0.71**	**0.24**	**0.29**

Table 4.7: Ability to differentiate the components allows MC-MATF model to outperform the other retrieval models.

	Short Queries			Long Queries		
Model	P	R	F_1	P	R	F_1
TF	0.68	0.51	0.58	0.66	0.52	0.58
MATF	0.79	0.60	0.68	0.72	0.55	0.62
MC-MATF	0.92	0.71	0.80	0.81	0.63	0.71

choices. Finally, we asked them if developers will benefit from looking at these code snippets. All the developers agreed that looking at these implementation choices will be useful. From this study, we conclude that identifying code snippets and showing the implementation choices can be very useful to developers. More details on this study including the summary of developer responses are available in Annexure 1 and on the jSense website[9]. Studies by Thummalapenta [100] and Nguyen et al. [101] show that reusable code snippets exist in abundance. Yang et al. [102] show that developers are using snippets from SO in GitHub projects. Hence, a tool like jSense is not only useful but also effective.

[9]http://jsense.epizy.com

4.5 Threats to Validity

Internal Validity We work with code snippets in SO which are relatively short compared to complete programs found in open source repositories such as GitHub. Structural comparison restricted to a method level as followed in our approach may not be effective in matching all topics especially associated with large programs. Given the millions of code snippets and topics discussed in SO, we find our approach useful despite its focus on such method-level comparison. It will be interesting to try our approach to extract code snippets from similar unstructured sources (Bugzilla, Jira, etc). These platforms may put forth extraction challenges. However, since our approach depends only on source code, and surrounding text describing the code, the approach by itself will not require any modification.

We have based our findings on our correlation of similarity metric (which is objective) with developer's perspective of heterogeneity (which is subjective). To reduce bias, we used three subjects who were experienced developers for experimentation. Our work is based on empirical evaluations with limited number of human subjects. It is possible that with a different set of developers, we might get different results.

Being a data-driven approach, availability of sufficient data becomes a pre-requisite for the approach to be successful. Multiple occurrences of the same snippet in SO discussions is necessary to increase confidence in the results for term frequency dependent techniques. Data in discussion forums will only grow over time and the same technique can be applied on other similar unstructured sources (bugzilla, for example).

External Validity There may be topics for which our results may not generalize to. To mitigate this risk, we have used topics of different types such as mathematical (factorial, HCF and LCM), language features (locking, string handling), data structures (circular counter, delete from array), algorithms from specialized domains (eigenfaces from image handling; linear regression is related to Machine Learning; chromatic number is from graph theory). The effectiveness of topics and challenges in building a useful topics list is also a bottleneck to the impact that this tool can make. We have used Mallet for topic modeling. Advances in topic models [103, 104] promise breakthroughs in solving this problem.

We have scoped our evaluation to Java language. Nothing in our approach except the parser is Java specific. Thresholds such as the similarity cut-off, aspect weight and the component weights may need to be changed for different programming languages. Given that index building for entire SO takes less than 2 hours, the entire process of empirical parameter estimation can be automated.

4.6 Related Work

Mining Examples Several researchers [53, 105, 106, 107] have discussed the importance of example-centric development. Blueprint [53] performs task oriented search for example code. XSnippet [108] focuses on searching for examples for the object instantiation task in Java. Portfolio [109] helps in visualizing dependencies. For a given source and destination type, PARSEWeb [110] goes beyond task oriented web search. It searches the web for method invocation schemes. Strathcona [93], MAPO [111] and Prospector [106] use structural details in code to get examples on API usages. In contrast, our work does not necessarily need to know about dependencies, input and output types, is neither task oriented nor API usage based.

Spotting Topics in Source Code Identifying source code behavior using NL has been researched upon under the name of Feature Location, Topic Modeling, Code Summarization, Code Description [75, 112, 113, 114] and Annotation. Topic models such as Latent Dirichlet Allocation has been applied [115] on source code to leverage the user defined vocabulary. Similar idea has been adapted in Structural semantic indexing [29] to leverage structural information. While the first technique limits to vocabulary, the latter uses call hierarchy information. We work on Java methods which are typically short snippets and have no call hierarchy information. Hence, these techniques cannot be used for intra-procedural structural search.

Structural Representations of Source Code Source code structure has been analyzed in the form of AST, CFG and PDG [116, 117]. Each representation operates at a different level of detail where certain syntactic or semantic elements are abstracted out. Higo et al. [118] show the relationship between functional similarity and three program attributes namely vocabulary, structural and method name similarity. However, in our case, we need an approach that finds structural similarity in the snippets so that it reveals heterogeneity in implementation. In our work, missing call hierarchy information and method level scope makes construction of value dependency graphs impossible. Further, we do not use input/output values or even execute the code samples. Our approach leverages light weight information retrieval and program analysis techniques.

Building Repositories Sourcerer [20] is closest to our work in constructing a code repository using IR techniques. Sourcerer does not group code snippets by topic which is essential for our purpose. Indexing big code has been important for code search [119] research. Our work leverages indexing techniques from such works so that several thousand code snippets can be searched near real-time with a sub-second response time. We have significantly improved the precision and recall of building a repository over the work of Vinayakarao et al. [37] by using an MCMATF retrieval model. We have added three key factors namely types, methods and recursion to our structural model.

Code Search Engines A simple web search on Google or Bing[10] results in popular websites that contain the vocabulary used in the query. Even the existing code search engines [83, 120, 77, 105, 121] such as searchcode [13] and Krugle [12] depend on the terms used in source code. Clues from the vocabulary used in the code may not be always useful in recognizing the code snippet [87]. Since there are no special processing on the structure of code snippets, a search for factorial in existing search engines does not return distinct implementation choices. We are interested in implementation choices and not just the best match by site reputation or text frequency.

[10]www.bing.com

4.7 Summary

jSense mines implementation choices for code snippets discussed in SO. Multi-component nature of posts in discussion forums demand usage of specialized retrieval models. We have adapted Multi-Aspect Term Frequency model and applied it for search on multi component text that includes source code as one of the components. To deal with source code as one of the components, we have proposed an improved selective set method which extracts structure from source code, and represents it as text. These models can be used for other retrieval tasks such as code completion and code comprehension. Hence, we have shared their implementation as well. These models and methods enable searching over partial programs of discussion forums with a precision of 92% and recall of 71%. With ever growing volume of discussion in these forums, the approach will become more effective in recognizing topics and mining code snippets that serve as implementation choices. Even though we mine implementation choices, this work is only a first step towards searching for code variants. As future work, we plan to extract the code context, automate the understanding of desired properties and apply this knowledge to compare the implementation choices, thereby retrieving variants. We also plan to apply our work on programming languages of the non-imperative kind. We look at improving the structural model to leverage the partial program analysis techniques so that the comparison of code snippets can be even more effective.

Chapter 5

Improving Code Search using Entity Retrieval

5.1 Introduction

Search has become indispensable in the modern programming context [24], where navigating through thousands of lines of code is infeasible. Developers search for code fragments or keywords to locate or navigate to a particular program concept, when they debug, write code, look for code fragments to reuse, or try to understand an API usage.

Current IDE search features (e.g., Eclipse search) or code search tools (e.g., Sourcerer [20], Portfolio [122]) work by indexing string tokens in the code as keywords. Therefore, searching for a particular programming concept requires an understanding of the syntactic equivalents of that concept. For example, if a developer wants to find whether her C code contains a function that uses an *integer array*, she may formulate a query based on the construct "`int files[]`". However, such a query will miss the occurrences of *functions* that have "`int *files`" in their definitions.

Currently, queries that include natural language (NL) give poor results [123]. Table 5.1 shows that this problem exists in established web-scale code search engines as well. This is because there is no mapping between the programming concepts and their associated syntactic forms in code.

We extract these mappings from developer discussions in SO. Programming concepts are identical in principle to *named entities* [124]. They are also named and may have multiple surface forms. First, to discover them, we infer the NL terms that refer to these entities by using Parts of Speech (PoS) tagging and pattern matching of these sequences. Then we extract the associated syntactic forms to create an entity knowledge base. Finally, each line of a given source code is annotated with their associated NL terms using the knowledge base, which then allows for regular keyword-based searches on these terms.

As a proof of concept, we have implemented our approach in a tool named ANNE[1]: ANNotation Engine, which includes entity knowledge bases for C and Java.

We evaluated the usefulness of the approach through a user study, in the context of Teaching Assistants (TA) providing feedback on submissions to class assignments. We recruited 16

[1]http://tools.pag.iiitd.edu.in:8092/anne/index.html

Table 5.1: P@10 of existing code search engines for NL queries containing programming concepts.

Query	Krugle [12]	openHUB [125]
declare array	0.1	0
concatenate two arrays	0.2	0
check if a string is a numeric type	0	0
assign to first element in an array	0	0

participants, who were currently or had been a TA for introductory CS courses. We used a within-subject study design, where participants in the Control condition used regular code, and those in the Experimental condition used code annotated with the programming concepts (as identified by ANNE). There were two tasks, one an assignment in C and the other in Java. We found correctness scores (precision) of participants to be equivalent across the treatment groups. This is likely because participants were TAs and had experience grading this type of submissions. The time to search was reduced by 29% without compromising on correctness and completeness.

Our key contribution is a technique to map lines of source code with relevant programming concepts, so as to support code search engines for NL queries. This allows the users to query on programming concepts using NL terms, and need not recall the exact syntactic terms or patterns.

5.2 Motivation

In this section, we present our formative study to motivate the work, followed by a discussion on potential applications, and a specific use case for this line of work.

5.2.1 Formative Study

As formative work, we surveyed to understand whether, and under what conditions developers use natural language terms in their search queries, and whether they face any difficulties when querying for a programming concept by using its syntactic form. We surveyed 25 developers working in leading software development organizations. 18 of these developers had 3 years or less experience in programming, while others had experience ranging from 3 to more than 10 years in industry. We consider the former group as novices. The survey questions and a summary of responses are listed in Table 5.2.

Our results indicate that novice developers, developers who are starting on a new project, or investigating code that they had not worked on for some time face difficulties in finding the right "code" in their project. 44% participants faced situations where the code did not match the NL term they would have used to search for that functionality. 72% participants felt that it is difficult to search for specific code syntax in current IDEs. One participant responded: "*I'd search for the patterns manually using a simple find operation.*" Another said, "*I might search for [different] keywords like 'mid' or 'pivot' when searching for a particular*

Table 5.2: Formative study on 25 real industry developers indicates that this research will be useful.

Survey Questions and Responses
(1) *Sometimes when reading a piece of code, the code snippet feels familiar but you may not know how it is popularly called. For example, in a very simplistic case you may be looking at a Quick Sort implementation. After reading the code, you may understand that the code is sorting integers, yet, you may not know that this algorithm is called "quick sort". Have you experienced this?* Yes: 44% (11); No: 56% (14);
(2) *Sometimes it is difficult to search for a specific code snippet in a project by using existing IDEs. For example, you may want to search how a particular element is initialized in the code. As another example, you may want to search for multiple substring computations and return statement with increment to find the Levenshtein distance implementation. Have you ever experienced this?* Yes: 72% (18); No: 28% (7);
(3) *Let us assume a situation where a developer wants to search for a quick sort implementation. The implementation is not available under the name "quicksort" and hence the developer wants to find all code snippets where "increment", and "mid-point computation" occur. To do so, the developer opens an IDE and creates a natural language query, "increment, mid-point computation". The IDE then automatically understands that, for Java, increment is "`++`" or "`+ 1`", and mid-point is "/2". It finds all methods where such constructs exist.*
(3.1) *How important do you consider this functionality? (>=3 on a scale of 5)* Important or Neutral: 92% (23); Not Important: 8% (2);
(3.2) *How often would you need to use this functionality? (>=3 on a scale of 5)* Often: 80% (20); Not so often: 20% (5);

sort algorithm". Unless the developers can find the correct query words, they will need to manually examine the code.

Code search still happens through keyword search. One participant noted "*[I] search the source code files using keywords/ Mnemonics in the hope that the developer might have used meaningful keywords in the code. Example: for the quicksort example, I might [search] for keywords like mid or pivot*". 60% felt it to be "important" or "very important" to support NL queries. For example, a participant mentioned: "*... [search function in an IDE] was highly useful, but it had no concept of natural language based result, which in my opinion would have proved to be even more useful and would have led to faster search results*".

Therefore, we conclude that programmers in the industry will benefit from a technique that maps source code lines with its associated programming concepts, which can be used to support search engines that need NL querying.

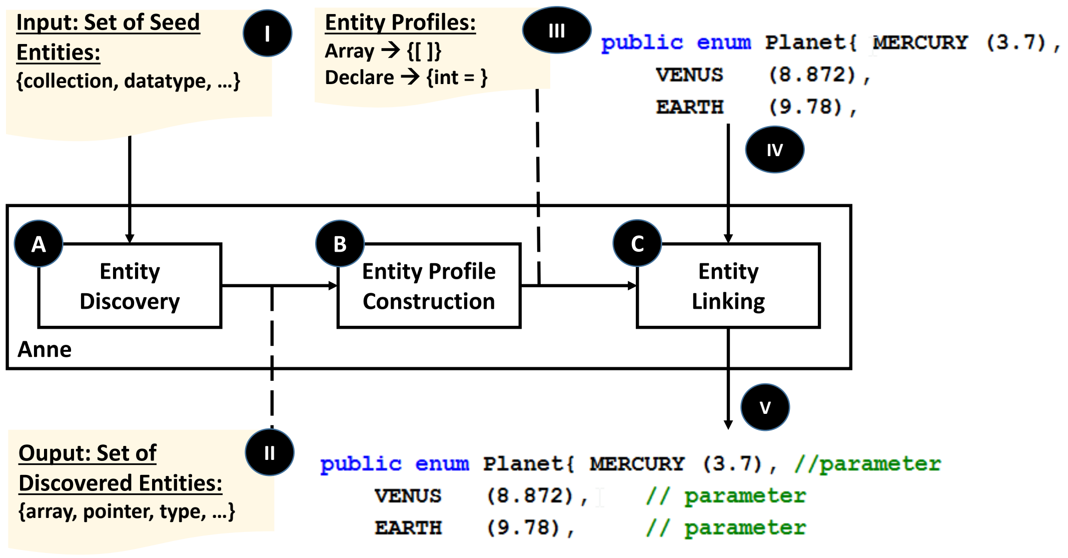

Input: Set of Seed Entities:
{collection, datatype, ...}
I
Entity Profiles:
Array → {[ ]}
Declare → {int = }
III
public enum Planet{ MERCURY (3.7),
VENUS (8.872),
EARTH (9.78),
IV
A
Entity Discovery
B
Entity Profile Construction
C
Entity Linking
Anne
V
Ouput: Set of Discovered Entities:
{array, pointer, type, ...}
II
public enum Planet{ MERCURY (3.7), //parameter
VENUS (8.872), // parameter
EARTH (9.78), // parameter

Figure 5.1: ANNE annotates input code snippets, line by line with natural language terms. These annotations help keyword based search engines to address NL queries.

5.2.2 Applications

In general, wherever there are NL queries involved in code search, our approach would be useful. In addition to web-scale code search, NL-based search over source code occurs in several software engineering contexts too, such as coding [24], maintenance [126], summarization [112, 127] and program comprehension [128]. To evaluate our work, we have used one scenario from academic code search related to programming assignments. Code search for programming assignments is another real problem and TAs find our tool very useful.

5.2.3 Problem Overview

Here, we list one use case and explain how our approach solves the problem.

Problem Sarah is a TA for an introductory Java programming class that has 100 students. She has to evaluate a programming assignment on ***enum*** which specifically requires students to use *parameterized enumeration*. To assess correctness, she has to look for the syntactic pattern of ***parameterized enums*** in all the 100 submissions. A keyword search on *enum* alone is insufficient as it will match all *enum* declarations (e.g., `enum { Mercury, Venus, ...}`). Note that *parametrized enum* takes the following form, `enum { Mercury(9,12), ...}`. Therefore, Sarah would either have to use regex in her query (such as `enum.*\{.*(.*);`) or search for occurrences of *enum*, and manually check each assignment to find the submissions that did not use *parameterized enum* properly. Even for experienced TAs, writing regex can be a challenging and error-prone task, especially when they have to discriminate between entities such as ***type casting*** and ***parameters***. ANNE takes NL input as *parameterized enum* and lists all occurrences and thus will help Sarah in this task.

Solution Figure 5.1 provides an overview of our solution to this problem. Our tool, Anne, mines SO posts to identify terms in SO questions that pertain to programming concepts, such as ***enum*** by using Parts of Speech (PoS) tagging. We refer to these programming concepts as *named entities.* More specifically, in Step A, when given a seed entity (e.g., ***array***), Anne identifies other entities (such as ***enum, arraylist, collection,*** etc.) through PoS tagging.

In step B, Anne analyzes the SO posts to mine associations between the entities and their syntactic patterns. In our scenario, Anne will create associations between the programming concept ***enum, parameter*** with its syntactic pattern: `enum { ( , ) }`. A result of this step, is a knowledge base of named entities and syntactical elements most associated with each entity.

Finally, in step C, given an input source code, Anne links the relevant programming concepts to each line of code. In our scenario, Anne tags the code lines from input source code, `enum {Mercury (9,12), Venus (10,13),...}`, with ***parameter*** as inline comment. As a result of this step, Sarah can now use keyword based search engines to locate the code using NL terms, such as ***parameter***.

5.3 Definitions

We define the terms that we use in the rest of the dissertation as follows.

Definition 1. *Let $c = (\tau_1, \ldots, \tau_n)$ be a code snippet where each τ_i is a syntactic token. Any subsequence p of c is a* ***Syntactic Pattern*** *occurring in c.*

Definition 2. ***Named Entities*** *in source code is a binary relation $E \subseteq P \times T$ such that $\forall e_i = (p_i, t_i) \in E$, t_i is a NL phrase used by developers repeatedly to identify an associated syntactic pattern p_i.*

We aim to map a given source code to its collection of entities. Hence, we first need to discover these entities. A bug description, a code comment, SO title, or any NL description of source code q, can be modeled as a sequence of terms or NL phrases $(t_1, \ldots, t_m)$ that points to a set of code snippets $\{c_1, \ldots, c_n\}$. Each code snippet c_i is a collection of lines $(l_{i_1}, \ldots, l_{i_o})$. From a large collection of such (q, c) pairs, our task is to find (p, t) pairs, where p is a syntactic pattern, and t is a term sequence from the vocabulary of q.

Definition 3. ***Entity Discovery*** *is the process of extracting candidate term sequences that represent entities in source code. In other words, we find the set of term sequences $T = t_1, \ldots, t_n$, such that each item t_i in it has at least one named entity (p_j, t_i) associated with it. We extract them from the available query-code pairs (q, c).*

Definition 4. *The* ***Entity Profile*** *is a mapping $\psi : T \rightarrow 2^P$ that takes term t as an input, and returns a set P' of syntactic patterns, $\{p_1, \ldots, p_k\}$ associated with t.*

Definition 5. ***Entity Linking*** *in source code is the process of associating a named entity e to a unit u of source code. In other words, it is a mapping $\Delta : C \rightarrow E$, where C is the set of source code units. In this dissertation, we use a line of code as a unit.*

Note that a line of source code can contain several syntactic patterns, and each syntactic pattern may be associated with several distinct named entities. Hence, the entity linking process may associate several entities to a line of code.

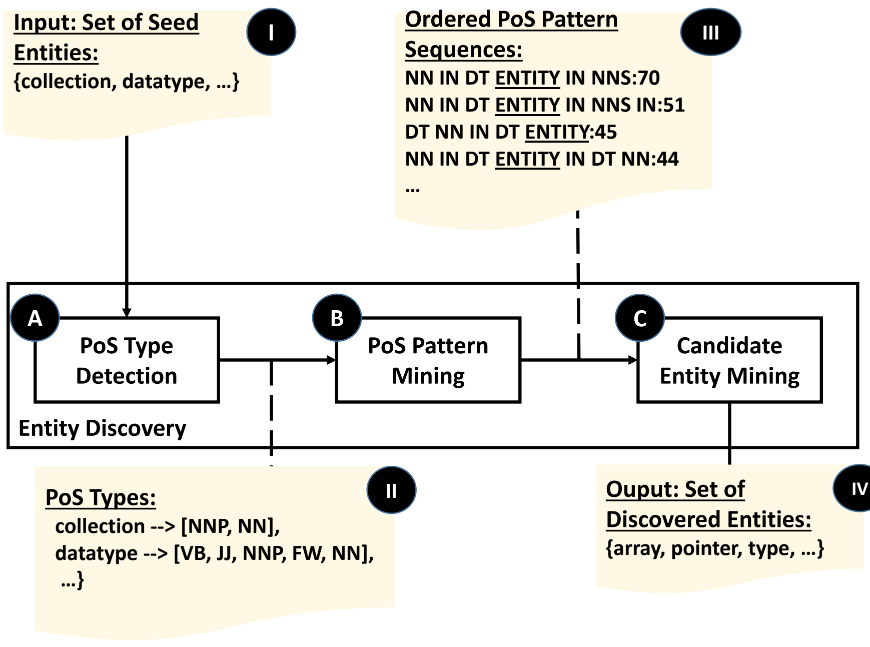

Input: Set of Seed Entities:
{collection, datatype, ...}
I
Ordered PoS Pattern Sequences:
NN IN DT ENTITY IN NNS:70
NN IN DT ENTITY IN NNS IN:51
DT NN IN DT ENTITY:45
NN IN DT ENTITY IN DT NN:44
...
III
A
PoS Type Detection
B
PoS Pattern Mining
C
Candidate Entity Mining
Entity Discovery
PoS Types:
collection --> [NNP, NN],
datatype --> [VB, JJ, NNP, FW, NN],
...}
II
Ouput: Set of Discovered Entities:
{array, pointer, type, ...}
IV

Figure 5.2: Entity discovery subsystem works on NL text using parts-of-speech (PoS) approach. We use seed entities to discover more entities that fit into the same grammatical sequence.

5.4 Approach

Our objective is to automatically tag lines of source code with their associated named entities. This is accomplished through three major steps: a) Entity Discovery, b) Entity Profile Construction, and c) Entity Linking.

5.4.1 Entity Discovery

We leverage PoS tagging to discover entities from SO titles. The intuition is that developers use similar sentence structures when they ask questions about a programming concept. For example, some of the SO titles are in the form of: *How to declare an **array** in Java?* and *How to declare a **list** in Java?* We exploit this similarity in sentence structures to extract entities. Figure 5.2 gives the workflow for this step.

We need to identify the entities relevant to a programming language. We select the seed entities from a popular tutorial site, Tutorialspoint[2]. Tutorialspoint groups topics related to the programming language into a short list. For example, Java had 39 topics and C had 29 topics. Some of these topics were phrases such as *type casting* which we mapped to a single word, *cast* in this case, with the help of a language expert. We use this list of processed topics as the seed entities (Figure 5.2 (I)).

For each seed entity, we extract and group the SO titles that contain that entity. Next, for each title in a group, we annotate its words with their corresponding PoS types (noun,

[2]We used the categories from http://www.tutorialspoint.com

Table 5.3: Patterns and frequencies for *conditional* in Java snippets found in SO.

Uni-gram Pattern	Normalized Frequency	n-gram Pattern	Normalized Frequency
`(`	1.00	`if ( ) {`	1.00
...		`( )`	0.50
`if`	0.65	`= ( ( ) )`	0.25
...		`( new ( ) {`	0.25
`while`	0.10	...	
`string`	0.06	...	

verbs, etc.,) by using an off-the-shelf PoS tagger (Stanford Log-linear PoS Tagger [129]). We extract all PoS sequences for every seed entity as shown in Figure 5.2 (II).

We use the position of the seed entity and its sequence to discover other entities. This is in accordance with research that has used PoS sequences to understand sentence structures [130]. Therefore, we identify the PoS sequence of each seed entity in all the SO titles in which they appear. For each seed entity, we rank the PoS sequences as per their frequency of occurrences (Figure 5.2(III)). For example, for ***array***, the most frequent pattern was NN IN DT ENTITY IN NNS where ENTITY is the placeholder for ***array***. As an example, the SO title, "How to determine type of object/NN in/IN an/DT array/ENTITY of/IN objects/NNS" has this frequent pattern. Same pattern appears in another title, "Get an array of int/NN from/IN a/DT string/ENTITY of/IN numbers/NNS". So we gather that both ***array*** and ***string*** have the same PoS sequence. Thus we discover more entities. To tune for precision, we use the top-most frequently occurring PoS sequence (Figure 5.2(C)). The output of this step is a list of discovered entity names (Figure 5.2(IV)).

5.4.2 Entity Profile Construction

Our goal in this step is to link the discovered entities with their syntactic patterns, to create a profile for each entity.

We leverage the fact that source code has repetitive syntactic patterns [42]. For instance, an ***array declaration*** has a syntactic structure composed of a few tokens, that are repeated across multiple source code snippets. Tu et al. [131] find that source code exhibits redundancies even in local context i.e., in short snippet of code being edited by a developer. They also show that frequency based n-gram patterns can be used to extract these redundancies. Further, Gabel and Su [132] find that the syntactic redundancy peaks at the line level. We leverage all these observations in finding repeating syntactic forms for entities at line-level using n-grams.

We intend to discover these patterns ($p_i \in P'$) in source code that are associated with specific entities (t) (e.g., ***array*** and ***conditional***). These pattern lengths ($|p_i|$) can vary. For example, an ***array*** has $|[\ ]| = 2$, but a ***conditional*** has $|if\ (\ )\ \{| = 4$. Let $SO_{\mathcal{L}}$ be the set of all SO posts tagged with a specific programming language $\mathcal{L}$ and containing at least one code snippet per post. We need to identify the most appropriate n-grams that represent a specific entity from $SO_{\mathcal{L}}$. We use the TF-IDF [133, 134] over n-grams to identify the syntactic patterns that are most associated with a given entity in $SO_{\mathcal{L}}$ (Table 5.3). To compute term frequency $tf(t, g)$ of an n-gram g, we use the $SO_{\mathcal{L}}$ posts containing the entity name in title. For IDF computation, we use all $SO_{\mathcal{L}}$ posts. Since SO post titles and code snippets are short in nature, we ignore the length normalization. Thus, we use the TF-IDF weight = $tf(t, g).log\frac{|D|}{df(g)}$ where $|D|$ is the total number of posts in $SO_{\mathcal{L}}$ and $df(g)$ is the number of such posts containing the n-gram, g. Table 5.6 shows the results of these steps for a few entities.

Controlling n-gram explosion We are interested in keywords related to programming concepts and not user defined terms (variable or identifier names). Hence, we collect and tokenize all code snippets from $SO_{\mathcal{L}}$, and rank distinct tokens (τ_i) by frequency ($tf(\tau_i)$). Tokens that are programming concepts will be ranked higher as opposed to user-defined terms, since fewer snippets would have overlap between usages of user defined terms in SO posts. So, we construct a list, $\phi(k) = \{(\tau_1, tf(\tau_1)), ..., (\tau_k, tf(\tau_k))\}$, of top-k tokens with highest frequency. The value of k needs to be large enough to contain all keywords of the programming language. We define a filter $F(\phi(k)) : c_i \rightarrow c_{if}$ which uses $\phi(k)$ to transform every line of code snippet c_i into a line c_{if} with only the top-k uni-grams. This reduces the total n-grams for each line of code, making the TF-IDF computation over n-grams tractable. Table 5.3 lists the n-gram associations that we mined using this approach for one entity, *conditional*.

In summary, in this step, for each entity that we discover, we identify its associated syntactic pattern, which we call the *entity profile*. The collection of these entities and their profiles serves as our entity profile knowledge base.

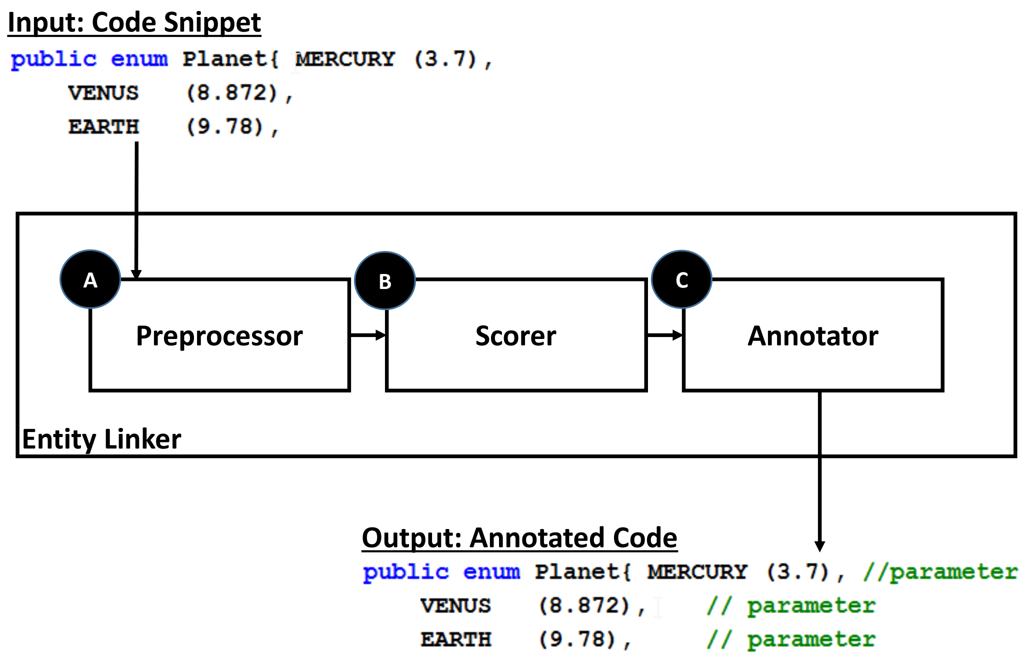
5.4.3 Entity Linking

Our goal in this step is to annotate every line in a given input source code snippet with entity names of those entities that appear in that line. We use the entity profile knowledge base for this purpose. Figure 5.3 gives an overview of this step.

We apply the same transformation $F(\phi(k))$ as in Section 5.4.2 to every line of input code (Figure 5.3(A)) to remove user defined terms, so that we get reduced number of tokens that pertain to programming concepts. We start with each term being a uni-gram and continue with cumulative aggregation into bi-grams, tri-grams, and so on, until all the n-grams are covered.

However, not all of these n-grams represent entities. For example `else ==` is not an entity. Therefore, we find the n-grams that actually represent entities by using the entity profile knowledge base. That is, we match the syntactic-patterns of an entity with that of the source code to determine if (and which) n-grams from the source code reflect surface forms of entities in source code.

Performing this matching is non-trivial, since n-grams are of different lengths. Therefore, one pattern can be subsumed within another. For example, the best syntactic match for *parameter* is the bi-gram `( )`, whereas the best match for a *conditional* is the four-gram `if (==)`. However, the bi-gram `( )` for *parameter* is subsumed by the four-gram `if (==)` for *conditional*. Therefore, if a line of code contains a *conditional* then both n-grams will match, where marking that line of code with both *parameter* and *conditional* is clearly wrong.



```
public enum Planet{ MERCURY (3.7),
    VENUS   (8.872),
    EARTH   (9.78),
```

```
public enum Planet{ MERCURY (3.7), //parameter
    VENUS   (8.872),     // parameter
    EARTH   (9.78),      // parameter
```

Figure 5.3: Entity linker subsystem works line by line on the input code, to find matching entity profiles. Entity names whose profiles match are stamped across the line, as shown in the example.

However, when we consider the code statement, `if (isTrue(...))`, both *parameter* and *conditional* entities exist. Consider the example shown in Figure 5.4(B)(line 6). ANNE stamps *parameter* along with a *loop* for this reason. We call this the subsumption problem.

To alleviate the subsumption problem, we use a scoring (Figure 5.3(B)) function. It creates a metric signifying how well an entity matches the line of code. In the first example, the longer n-gram is a better match (*conditional* vs. *parameter*). However, shorter n-grams are also of interest, as we see in the second example. Therefore, instead of making the entity assignment a binary decision, we rank the n-grams using the scoring function (Equation 5.1).

$$Score = \delta w_u + (1 - \delta) w_n \tag{5.1}$$

We use weights for the uni-gram and n-grams based on their TF-IDF values. Table 5.3 shows the normalized weights for an entity. Since we need to balance between the uni-gram weights (w_u) and n-gram weights (w_n), we empirically identify the distribution factor δ to be 0.6 as this gives us the best results.

Once we determine an entity for a line of code, the annotator tags the line with that entity as an in-line comment (Figure 5.3(C)). This allows regular keyword-based searches to search on the entities. Note, a line of code can have multiple relevant entities. However, too many entities per line can reduce the quality of the search, as well as cause readability issues if the end user wishes to look at the entities. In the assignments that we use for our user study, we did not find a line of code with more than four entities, therefore, we use that as our threshold. We leave finding the optimal number of entities per line of code as future work.

5.5 Evaluation

Our evaluation goal is twofold: a) *How well does Anne link entity names to lines of code snippets?* and b) *How useful are these annotations*? However, due to the multistage nature of the entity linking process, we divide the first goal into two sub-goals and address them in this section. We address the second question by conducting a user study, which we present in Section 5.6.

```
public enum Planet{//enum,parameter
   MERCURY (3.7),  //parameter
   VENUS (8.872),// parameter
   EARTH (9.78), // parameter
   MARS (3.7),    // parameter
   JUPITER (24.79),// parameter
   SATURN (10.44),// parameter
   URANUS (8.87), // parameter
   NEPTUNE (11.15);// parameter
    final double surfaceGravity;

   Planet (double
       surfaceGravity){// parameter
      this.surfaceGravity =
       surfaceGravity;
   }
    ...
```

(A) Enum Task

```
int main() {
    int N,A;
    scanf("%d%d",&N,&A);// parameter
    int arr[N];              // array
    int i, left, right, flag=0,sum;
    for (i=0;i<N;i++) {    // loop,parameter
      scanf("%d", &arr[i]);}//array,parameter
    left=0;
    right=N-1; //decrement
    while (left!=right){ // parameter
      sum=arr[left]+arr[right]; // array
      if (sum<A)      // parameter,
       increment,decrement
        left++;         // increment
      else if (sum>A) // parameter
        right--;        // decrement
   ...
```

(B) IncDec Task

Figure 5.4: Tagged versions of the tasks (A) Enum and (B) IncDec that were provided to the participants in the user study.

5.5.1 Entity Discovery

The first question we need to answer to fulfill our goal is: *How well can we automatically identify entities that represent programming terms from SO titles?*

We automatically discover entities based on the position of the seed entity in the PoS sequence in SO titles. Therefore, it is possible that some of the terms that we identify as entities are incorrect. The discovery of entities depends on the length of the SO titles and the PoS sequence lengths.

Of the 0.9 million posts in our dataset, 639K were questions in Java, whose answers also contained source code snippets. The median number of terms contained in these titles was 7. Similarly, we had 139K titles in C, associated with answers containing source code snippets; the median for number of terms in titles was 6.

Table 5.4: No. of entities discovered is related to the length of PoS patterns considered in our approach. Longer patterns produce fewer entities that exhibit higher level of similarity to seed entity.

#PoSTerms	#Entities	#PoSTerms	#Entities
5	10k	7	12k
6	22k	8	6k

Table 5.5: Performance of ANNE Entity Discovery module. Experiments were carried out on a gold set with 1:1 noise and 1:3 noise. F_1 indicates the F_1-score and —E— stands for the number of entities discovered.

P_f	Java 1:1		Java 1:3		C 1:1		C 1:3	
	F_1	$\|E\|$	F_1	$\|E\|$	F_1	$\|E\|$	F_1	$\|E\|$
2	0.76	21K	0.14	35K	0.66	12K	0.71	20K
3	0.94	5K	0.87	9K	0.74	10K	0.64	17K
4	0.82	7K	0.87	12K	0.81	6K	0.71	11K
5	0.82	5K	0.76	7K	0.87	3K	0.86	5K
6	0.94	2K	0.91	3K	0.89	1K	0.91	2K
7	0.81	696	0.78	1K	0.86	751	0.86	1K
8	0.81	524	0.78	807	0.89	376	0.89	606
9	0.51	99	0.47	136	0.77	253	0.71	402

Next we perform a sensitivity analysis using the PoS sequences around the median values and the number of entities generated. More specifically, we evaluate PoS sequence lengths of 5, 6, 7, and 8 in our dataset. Our findings are presented in Table 5.4.

For each sets of entities discovered, we calculated the precision of results. True positives were manually evaluated by the first two graduate students who were experienced in Java and C. They verified that: (1) the entity name appeared in a Java [135] or a C [136] textbook as a term related to a programming concept or a programmatic structure, and (2) the discovered entity had a syntactic pattern. They individually identified the true positives and compared their results. Any differences were discussed until they both agreed about a term. If there was disagreement that could not be resolved, then that term was dropped from the list.

From a random sample of SO titles, two experts manually extract the first 30 entities. The first 25 entities that they agreed on (i.e., the intersection of their results) is used to build a *goldset*. Stemming gives the root of the words and thus helps in precision. Classifier separates titles with seeds from the rest. Mixer adds noise in required proportion. Our goldset consists of all SO posts containing these 25 entities (162K posts) and thrice (a 1:3 split) that much of noise (i.e., other posts that do not contain any of these 25 entities). To compute recall, we run ANNE with five of these 25 entities as seeds. Table 5.5 gives the F-measure and count of entities discovered. We report these values for both 1:1 and 1:3 splits. Notice that with increase in noise, the F-measure drops. We observe recall of 0.91 for both Java and C, at a proximity of 6.

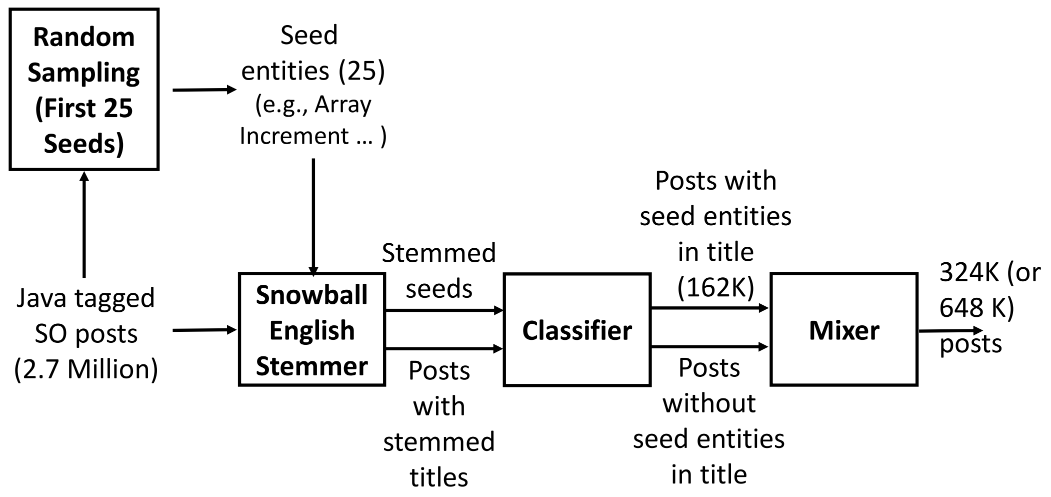

Random Sampling (First 25 Seeds)
Seed entities (25) (e.g., Array Increment …)
Java tagged SO posts (2.7 Million)
Snowball English Stemmer
Stemmed seeds
Posts with stemmed titles
Classifier
Posts with seed entities in title (162K)
Posts without seed entities in title
Mixer
324K (or 648 K) posts

Figure 5.5: The goldset for evaluation is created from SO posts by mixing posts that contain seed entities in the title with those that do not have them.

Next, we use ***array*** as the seed entity for evaluation for both Java and C on the entire SO corpus. We discovered 20 additional entities for Java, and 18 for C, which resulted in a precision of 0.78 for Java when considering a PoS sequence of 7 terms. For C, it gave a precision of 0.77 when considering a PoS sequence of 6 terms.

Closest to our work are the Named Entity Recognizers (NER). Stanford NER [137] is a popular implementation of linear chain Conditional Random Field (CRF) sequence models. Our approach is much simpler heuristic-based approach which does not need training data. Yet, in principle, this can be modeled as a 2-class classification problem. We trained it with 10K tokens with POS tags where each SO title is a document. Trained NER models that we built are shared on Anne website. We get near-zero precision and recall of 0.2 on our goldset with these models. The objective for this work is to showcase that entities can be detected and are useful for search. Hence, we do not focus on improving the training data or finding features for the classifier.

5.5.2 Entity Profile Construction

The evaluation question that we ask here is: *How well can we map entities to their syntactic patterns?*

For each entity, we calculate the precision of the syntactic patterns (a n-gram sequence) extracted in the entity profile construction stage. That is, we evaluate our pattern recommendation for an entity. To do this, we can analyze the top-1, top-2,..., top-n pattern recommendations for each entity. Note, when we consider top-k recommendations, the order in which a pattern appears does not matter, since “all” these k-patterns are linked to the entity.

We analyzed top-1 to top-8 patterns, and found that the best precision is at top-4. This is likely, because if we have too few recommended patterns, then we miss entities. However, if there are too many recommended patterns, it adds noise to the process. Therefore, we assess our recommendation by computing precision@4 (p@4) [133].

Table 5.6: Manually computed precision@4 and the top pattern discovered for some of the entities. We use top four patterns while annotating source code.

Entity	p@4	Pattern	Entity	p@4	Pattern
array	1.00	[ ]	*conditional*	0.75	if ( ) {
decrement	0.75	- -	*increment*	0.75	+ +
loop	0.50	for (; ;)	*parameter*	0.75	( )
pointer	0.50	int *	*variable*	0.75	int

Table 5.7: Two factor design that counterbalances the treatment and the task.

	Enum	IncDec
Tagged	Group 1	Group 2
Untagged	Group 2	Group 1

The precision of the entity profile knowledge base depends on the richness and the volume of our data. In SO titles for C, the entity ***array*** appeared for more than 14K times. Because of this, ANNE gets perfect precision (Table 5.6). However, although some entities, such as ***pointers*** had more than 10K occurrences, they had many associated patterns: `struct *`, `int *`. This leads to lower precision. We evaluate the patterns for the eight entities that we found in the user study tasks. Table 5.6 gives the p@4 for these entities, and shows the top pattern. The average precision for these entities is 0.72. We also compute mean reciprocal rank (MRR). MRR is computed as: $MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$ where N is the number of entities, and $rank_i$ is the rank of first relevant pattern for the i^{th} entity. MRR across Java and C for the entity profiles turns out to be **0.71**. ANNE loses on longer patterns and gains on shorter patterns primarily because of subsumption.

5.6 User Study

We evaluate the usefulness of ANNE through a user study. We recruited 16 participants who were currently a TA or had been one for programming courses. All participants had similar background and programming language skills. They were given two real programming assignments to grade from classes at a lead university. One assignment was from the class taught using C, and the other assignment was from a class taught using Java.

Each submission for these assignments was annotated with the associated named entities by using the entity profile knowledge base, which was created by using the September 2015 SO dump. We implemented a simple search tool (downloadable from ANNE website) to serve as a testbed to evaluate the usefulness of ANNE in a controlled environment.

5.6.1 Study Design

We selected the tasks for the study by first analyzing all the assignments from the two classes. We focused on assignments given earlier in the semester as these were likely to be easy to

Table 5.8: Descriptive statistics of number of incorrect assignments found by participants.

	Enum (27 Incorrect)		IncDec (15 Incorrect)	
	Mean	Median	Mean	Median
Tagged	24.63	24.00	13.88	14.00
Untagged	22.38	25.50	12.00	13.00

Table 5.9: Terms used to calculate correctness and completeness scores for a submission S.

Term	Description
True Positive (tp)	S correctly classified as incorrect.
False Positive (fp)	S wrongly classified as incorrect.
False Negative (fn)	S wrongly classified as correct.
True Negative (tn)	S correctly classified as correct.

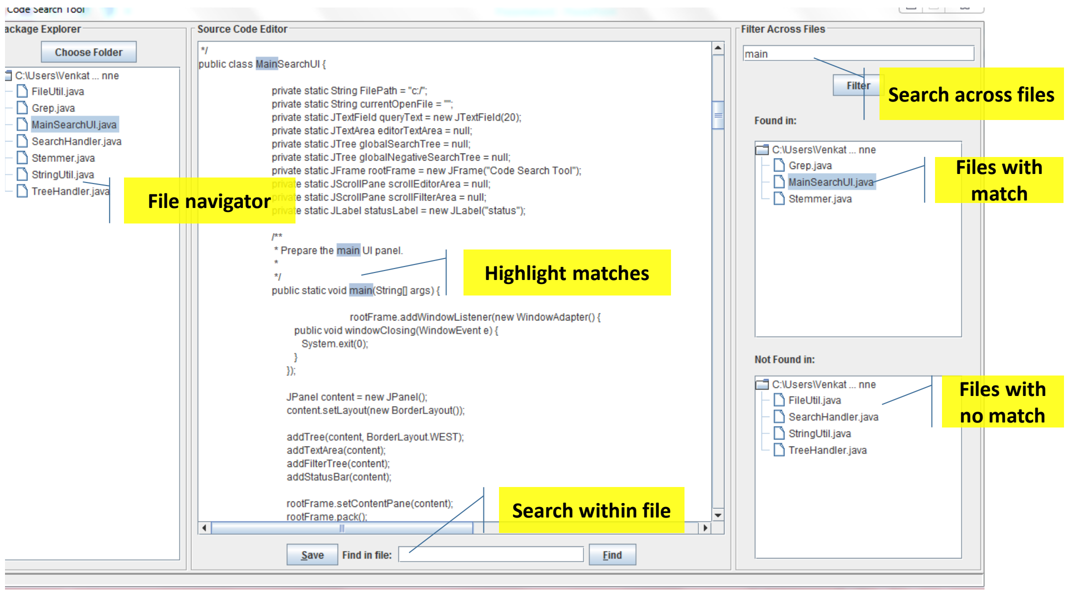
evaluate. We needed the tasks in our study to be within 20 minutes, so as to allow the study to be completed in an hour. We performed a pilot study with three graduate students to identify the tasks to be used for the study. For the pilot, we randomly identified six assignments (3 from each class) and their student submissions. Based on our pilot studies, we selected the following two assignments, since the pilot participants could easily understand the code of these two assignments, and took about 15-20 minutes to complete the task.

The first assignment (referred to as *Enum*) expected students to use parameterized enumerators when calculating the weight of a person on different planets. The second assignment (referred to as *IncDec*) asked students to operate over a sorted list, while ensuring that their algorithm had a time complexity of O(n). The former assignment was in Java and had 73 student submissions; the latter was in C and included 96 submissions. Figure 5.4 provides snippets of a student submission for both tasks. Participants had to evaluate the correctness of each student submission and stamp their feedback on the incorrect ones. The Enum and IncDec tasks had 27 and 15 incorrect submissions, respectively.

We followed a two-factor, within-subject study design. We created untagged and tagged versions for each set of submissions, where the former was used as the control condition, and the latter as the experimental condition. We counterbalanced the order in which participants were placed in a treatment group, as well as the task-order that was associated with a specific treatment (Table 5.7). That is, eight participants had to evaluate assignment submissions that were untagged as their first task, while the other eight participants evaluated the tagged submissions as their first task. Similarly, half the times Enum appeared as a tagged version, and as untagged for the rest.

Participants first filled out background information, and were provided a tutorial of the tool (10 min). They were then asked to evaluate a sample assignment (on pointer usage) to gain a hands-on understanding of the tool and the evaluation that they had to perform (10-15 min).

They were given instruction sheets that explained the different features of ANNE (see Figure 5.6 for a snapshot of UI). This ensured that they spent their time on the task and not on learning the tool. They were also provided with instructions on how to evaluate



```
*/
public class MainSearchUI {

            private static String FilePath = "c:/";
            private static String currentOpenFile = "";
            private static JTextField queryText = new JTextField(20);
            private static JTextArea editorTextArea = null;
            private static JTree globalSearchTree = null;
            private static JTree globalNegativeSearchTree = null;
            private static JFrame rootFrame = new JFrame("Code Search Tool");
            private static JScrollPane scrollEditorArea = null;
            private static JScrollPane scrollFilterArea = null;
            private static JLabel statusLabel = new JLabel("status");

            /**
             * Prepare the main UI panel.
             *
             */
            public static void main(String[] args) {

                        rootFrame.addWindowListener(new WindowAdapter() {
               public void windowClosing(WindowEvent e) {
                 System.exit(0);
               }
              });

              JPanel content = new JPanel();
              content.setLayout(new BorderLayout());

              addTree(content, BorderLayout.WEST);
              addTextArea(content);
              addFilterTree(content);
              addStatusBar(content);

              rootFrame.setContentPane(content);
              rootFrame.pack();
```

Figure 5.6: Code search tool used for giving feedback to student assignments. This tool allows us to toggle tagging on and off for evaluation.

submissions, which included the problem statement of the assignment, an explanation of the expected answer, and the feedback that needed to be stamped on the submission. We used the instructions that were provided to the TAs of the (actual) classes to create these materials.

Once they were comfortable with the tool, the experiment started. The time for each task was fixed at 20 minutes. We conducted an exit interview, where we asked whether they would use ANNE for the next class that they TA for. Resources used in this study including video recordings of the study are available on the ANNE website.

5.6.2 Results

Subjects in Experimental condition (S_{tag}) heavily used NL terms in their queries. For example, for the IncDec task, one participant (P17), by using a single search query "increment decrement", was able to identify all the incorrect submissions. Some participants (e.g., P13) created more queries: "increment; decrement; increment decrement; feedback" to get the same results. In contrast, subjects in the Control condition (S_{raw}) made more sophisticated queries, many of which were not successful. For example, P6 tried many queries: "left;++;++ --;binary;while <;for", and was only able to find 13 out of the 15 incorrect submissions.

Table 5.8 provides the mean and median number of the "incorrect submissions" that participants found. Note that Enum had 27 incorrect submissions out of the 73 total submissions, and IncDec had 15 incorrect submissions out of the 96 total submissions.

We evaluate the quality of the participants' work by calculating the completeness and correctness metrics, for each task (Enum vs. IncDec) and treatment (tagged vs. untagged). Table 5.9 lists the terms used for these metrics. The correctness metric is calculated as the number of correct classifications divided by the total number of classifications (tp / (tp +

Table 5.10: Correctness/Completeness metrics of participants with std. deviation in parentheses.

	Correctness		Completeness	
	Tagged	Untagged	Tagged	Untagged
Enum	1.00(0.00)	1.00(0.00)	0.91(0.25)	0.83(0.06)
IncDec	0.98(2.79)	0.98(2.59)	0.93(0.20)	0.80(0.09)

Table 5.11: Time taken to complete (in minutes) assessment for tagged and untagged versions.

	Enum			IncDec		
	Mean	Median	SD	Mean	Median	SD
Tagged	8.71	7.90	2.61	11.50	13.88	4.66
Untagged	12.90	14.78	3.43	12.98	12.80	4.12

fp)). The completeness metric is calculated as the number of true positives divided by the total number of true positives and false negatives (tp / (tp + fn)).

Table 5.10 provides the correctness and completeness metrics along with standard deviations. We observe that both S_{tag} and S_{raw} obtained very similar correctness scores. This likely occurred because after identifying the submissions, participants evaluated the code before stamping their feedback. Since they were previously TAs and the assignment was relatively simple, their evaluations were accurate. However, for the completeness metric we see that S_{tag} perform better than S_{raw}. This means that more incorrect submissions were missed by S_{raw}. The experience and individual differences play a larger role when participants use a keyword-based search, explaining the higher variance in S_{tag}.

Next, we analyze the time to complete the task. We found that a majority of participants in both treatments followed a two-stage process. As a first pass, participants used the search feature to locate those submissions that *did not* contain the terms in which they were interested. Then as a second pass, they manually investigated the submissions to double check their work[3] if they had time.

Here we report on times to complete the task of the first phase alone, since this best compares the two search processes. Table 5.11 reports the time (in min) to complete the task and the standard deviation. We observe that S_{tag} was faster. There is a bigger time difference for the Enum task as compared to the IncDec task. This is likely because evaluating the IncDec task was more complex, since participants analyzed the algorithm to determine its complexity. In the Experimental treatment, participants more quickly obtained the set of incorrect submissions, therefore, they may have spent less time in evaluating the other correct submissions.

Next we test for statistical difference between the two treatments for the completeness and time metrics. We perform Shapiro-Wilk test of normality (at $p < 0.05$) and find that

[3]We report correctness and completeness metrics after they finished their tasks and, therefore, include both passes.

both time and completeness are normally distributed. Therefore, we use two-way ANOVA to account for any interaction effects between task and type. For completeness, we find no statistical significance at $p < 0.05$ level; F(1,29)=3.15, p=0.08. S_{tag} had higher completeness metrics. There was no interaction between the treatment (tagged vs. untagged) and the task; F(1,29)=0.018; p=0.89. When using Cohen's d, we get an effect size of 0.64 (medium). So, we gather that tagging does not negatively affect completeness significantly.

When considering time, we see a significant difference ($p < 0.05$) between the two treatment conditions; F(1,29)=4.50, p=0.04. S_{tag} took less time to complete tasks. There was no interaction between the treatment and task; F(1,29)=1.15; p=0.29. We get a Cohen's d value of 0.75 (medium).

In summary, participants are able to complete the tasks in much shorter time (29% less) without compromising significantly on correctness and completeness. The post-task interviews show that tagged search is useful.

5.7 Limitations and Threats

Our approach will be even more useful if we can discover entities with multiple terms in their name, and also by detecting patterns across multiple lines of code. In principle, our approach can still be used where we treat every snippet as one single line and apply the same technique as we did for uni-gram entity names. For snippets with longer lines of code, the number of n-grams in such long lines increase and this causes computational overhead. Hence, we look forward to work on more efficient models to support this research.

Handling the subsumption problem where shorter patterns appear inside longer patterns is very hard to address in a language agnostic manner. Although our scoring function alleviates this issue, this can still impact correctness and hence needs attention in future work.

Users may find it unintuitive to formulate queries with terms that do not appear in code. In our case, we tagged the code with terms and thus we circumvented the issue.

We have limited the implementation and evaluation of our work to Java and C programming languages. Yet, other languages, especially, markup languages and functional languages may put forth different challenges. While our technique is statistical by nature and leverages Information Retrieval techniques, the implementation uses language dependent techniques for parsing, and to clean up the snippets.

5.8 Summary

In this work, we present a technique that leverages the structural similarities in how people phrase programming questions, and the repetition of syntactic structures in source code, to map source code lines to their programming concepts. This opens up new opportunities to support tools and techniques that connect natural language to source code. Search engines and IDEs can use this mapping to improve code search over NL queries. We show how such a mapping can help in academic assignment search through our tool prototype, ANNE.

Even though our approach, at least in theory, can be extended to program blocks with multiple lines of code, it might need more sophisticated code models for syntactic matching.

Another interesting research direction will be to support longer NL phrases. Our approach is predominantly language independent. We look forward to thoroughly evaluating our work on a variety of languages, in addition to Java and C, especially in functional and scripting languages.

Chapter 6

Reducing the Parsing Problems in Stack Overflow

6.1 Introduction

With the growth in popularity of Q&A websites, they have become an attractive data source for crowd knowledge on code snippets. Code snippets discussed in forums such as SO have been used extensively in software engineering research [138, 139, 140]. Yang et al. [138] extracted SO code snippets to investigate and understand the extent to which the snippets obtained from SO are reused in GitHub projects. Yang and Hussain [139] used these snippets to provide a study of reusable code of four popular programming languages namely C#, Java, JavaScript and Python. Subramanian and Holmes [140] performed snippet analysis to extract structural information from short plain-text snippets that are often found in SO.

While several such research efforts depend on parsing the code snippets extracted from SO, a majority of the code snippets are not compilable or even parseable [141, 139] as they are partial programs presented for discussion purpose. For example consider the code snippet listed in Listing 6.1. It suffers from a parsing issue due to embedded HTML elements (&lt; and &gt;). We posit that, with the knowledge of various kinds of such parsing issues, these partial programs can be curated without changing their semantics. As we reduce the parsing problems, more snippets become parseable thereby allowing the researchers to benefit more from SO data.

More than source code extraction, curation has been of interest to the research community [30, 142, 143]. Dagenais and Hendren [30] propose a framework for type inference and resolving ambiguities in partial programs. GRAPA tool [143] focuses on guessing the types so that code completion tools are benefited. Vesperin [142] curates code snippets using the other snippets in the same discussion. These tools largely ignore the fundamental problem of extracting code snippets from large dumps with heterogeneous data such as discussion forums and parsing them to generate the Abstract Syntax Tree (AST). Instead of focusing on type inferences and resolving ambiguities, we focus on the extraction part of the problem. We scope our work to understanding the dominant parsing issues. To the best of our knowledge, this is the first systematic study to characterize the parsing issues in this context.

In this work, we ask the following Research Question (RQ):

```
public void add(K key, V val)
 {
   Collection&lt;V&gt; c = get(key);
   if (c == null)
     put(key, c = createCollection());
   c.add(val);
 }
```

Listing 6.1: An erroneous snippet in SO.

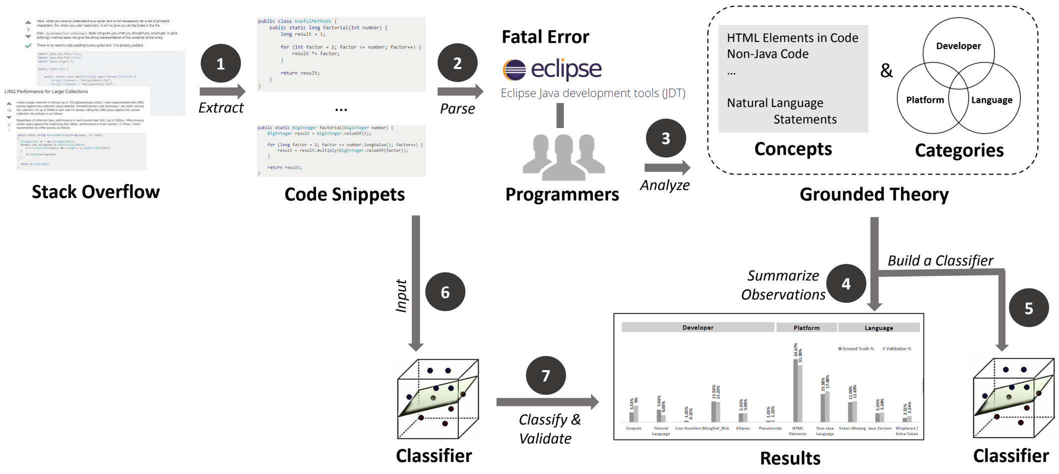


Figure 6.1: Overview of our study of variants. (1) We extract code snippets from SO. (2) We attempt to parse those snippets using eclipse JDT. (3) For snippets that do not parse, we analyze using a GT approach. (4) We propose a list of issues that dominate while parsing. (5) We build a classifier using a training set from our study. (6) We apply this classifier on entire SO data. (6) Thus, we validate our theory on entire SO.

RQ: What are the dominant issues plaguing Java code snippets from SO that extraction tools must address to make them parseable code snippets?

Our study involves seven major steps: (1) We extract all the text in the “code” markup in posts tagged as Java are java code snippets. (2) We attempt to parse these snippets using a Java compiler. (3) We analyze the erroneous snippets manually. Three programmers analyze the results using the GT approach. We arrive at concepts, sub-categories and categories. (4) We report the dominant issues causing parsing problems as the results of our study. (5) We validate our findings using manual inspections. As a proof of concept that we can automate these concepts, we use the snippets that we manually analyzed for building a classifier for three of the concepts. This classifier takes as input a code snippet and outputs the list of issues that may be causing parsing problems. (6) Since, our manual analysis in step 3 was on limited snippets, we use this classifier to discover issues on all the Java snippets in SO. (7) We summarize our observations and compare them with the results obtained from manual GT study. We show that our theory holds and list down the dominant issues in parsing.

Based on this knowledge, we build a tool named jMechanic which increases the number of clean snippets. Figure 6.1 gives an overview of our work.

Rest of the chapter is organized as follows. Section 6.2 gives the background on code snippet parsing and relevant tools along with the relevant terminology. In Section 6.3, we outline the Grounded Theory approach. Section 6.4 describes our data analysis with examples we observed in the study. Section 6.8 answers our RQ on the causal factors behind the dominant parsing issues. It provides the distribution of these issues in SO. The design and implementation details of the classifier we used to validate our findings are in Section 6.7. We discuss the threats to validity in Section 6.10 and related work in Section 6.9.

6.2 Background and Terminology

For most research projects [144, 145, 146] where researchers are interested in Java code snippets from SO, they extract the posts tagged as "Java" from the SO dump[1]. Such posts are assumed to contain Java code snippets. This extraction is typically a simple step of parsing the "`<code></code>`" markup which is used to encase non-textual content in SO posts. Once the code snippets are extracted from the dump using any XML parser, they parse the code snippets using a Java compiler. One popular implementation of Java compiler is the Eclipse Java Development Tools[2] (JDT). It provides users the ability to construct an AST from snippets. Additionally, in the event of failure in compilation, it presents developers with errors that caused the failure. This functionality is provided by its IProblem (org.eclipse.jdt.core.compiler) interface. Eclipse JDT produces an AST representation of code snippet whenever possible along with the compilation errors if any. In principle, we assume that a code snippet is *erroneous* if there is even one IProblem of type "Error". To avoid confusions over compilation and parsing errors, we use the term *unclean* to refer to such erroneous snippets that do not produce an AST. For example, the unclean snippet shown in Listing 6.1 produces the error "*Syntax error on token(s), misplaced construct(s)*". The error is due to the presence of &gt; and &lt; in code. As another example, the following code produces the error, "*Syntax error, varargs are only available if source level is 1.5*" due to the compiler version not supporting newer features:

```
public static void main (String... args) {

    for (Char char : chars) {
        System.out.println(char);
    }
    int k = 0;
    for (int m = 0; m != M; m++) {
        if (flags[m] != 'B') {
            swap(flags, k++, m);
        }
    }
    for (Char char : chars) {
        System.out.println(char);
    }
}
```

[1]https://archive.org/details/stackexchange
[2]www.eclipse.org/jdt/

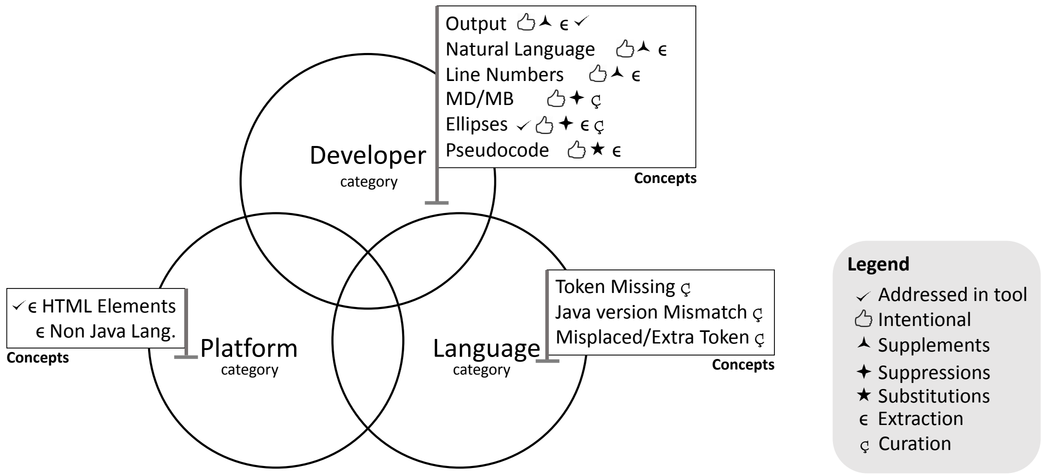


Figure 6.2: Discovered concepts and categories. Concepts in the Developer category are further classified into the subcategories of Supplements, Suppresions and Substitutions. Some of the concepts are introduced intentionally by developers for the purposes of brevity and focus. These are marked as “intentional” in this figure. Our focus is on extraction related issues instead of curation issues. We have distinguished them here. We build a classifier for three of these concepts namely Outputs, HTML Elements and Ellipses. We show that this helps jMechanic tool to improves on its ability to parse more snippets.

A newer version of Eclipse JDT would not throw this error. Since, snippet consumers may not give attention to error details, we have included this class of errors also in our study.

Researchers have largely ignored the unclean code snippets. We focus on reducing serious and fatal parsing errors of these unclean snippets. Instead of developing a tool for curation, our focus is on studying these erroneous snippets.

6.3 Research Method

We have adopted the Grounded Theory (GT) [147, 148] approach to address the research questions. GT helps the researchers in uncovering common patterns through continuous data observation. It is based on the idea that while collecting data and simultaneously analyzing the data, a theory can be further developed or newly created. In this case, it aims to discover the way developers write partial programs in SO.

GT consists of five steps. First, all the data is collected from the source. Second, data is analyzed and labeled using a code that summarizes its meaning. Third, after assigning discrete codes to the data, these codes are then grouped into concepts. Next, the relationships between concepts are examined and classified into categories. We continue the labeling activity until we reach a saturation. Finally, based upon these categories, a new theory is proposed.

6.4 Data Analysis

We used the error message obtained through the IProblem interface of Eclipse JDT as our initial labels. On manual inspection of the snippet, the error message and the post containing the snippet, we associate a concept to each unclean snippet. Following the GT approach to discover concepts, we chose the size of each sample set to be 50 randomly selected SO posts. We refined the concepts until we reached saturation after 3 iterations (150 posts).

Our study shows that there are three major categories: 1) Developer induced issues, 2) Language specific issues, and 3) Platform (SO) induced issues. We identified 11 different concepts from these aforementioned categories. Figure 6.2 shows these categories and concepts.

6.4.1 Issues due to Developer Behavior

Code snippets posted by developers on discussion forums are used to enhance discussions. Hence, we refer to them as "intentional" changes introduced in code snippets by developers in Figure 6.2. A code snippet becomes unclean because the developer either added something to the code snippet (Supplement), removed some tokens from the code (Suppression) or substituted tokens with non-Java constructs (Substitution) from the snippets. We group the concepts produced by GT under the aforementioned sub-categories.

Supplement to Code Code snippets are annotated with additional information of various kinds. Although developers extensively use Natural Language (NL) statements around the source code outside of the "`<code></code>`" block, they still find a need to adulterate the code with useful content to make the discussion much more easier. Following are the concepts that we discovered related to this sub-category.

Outputs Developers also add outputs from compilers and parsers to their code snippet to provide greater context. Examples of such outputs are Stack Traces, Error Messages or the result of an execution. Outputs form a special case of non-Java text constructs that assist debugging or to better represent how various subroutines work together during execution. Hence, they show up often in SO. Therefore, we chose to keep a concept dedicated to this kind of content. Following example occurs in a post [149].

```
>Exception in thread "main" java.lang.NullPointer
Exception
  >>at org.lwjgl.opengl.GL11.glMatrixMode(GL11.java:
2052)
  >>>at game.engine.GameLoop.start(GameLoop.java:22)
  >>>>at game.engine.GameLoop.main(GameLoop.java:15)
```

Natural Language The code snippet below [150] presents an example of developers interleaving Natural Language into their code. In an SO post [150], we find the following block of text, "...Your Implementation of PieChart goes here..." which the Java parser cannot recognize.

```
package com.mypackage;

public class PieChart extends View {
    ...Your Implementation of PieChart goes here...
}
```

Line Numbers Line Numbers are supplements used to specify a particular sequence of characters. The addition of line numbers to snippets presents an interesting problem. Consider the first snippet [151] shown below.

```
1 while (true) {
2   int nBytesRead = src.read(buff);
3   if (nBytesRead < 0) {
4     break;
5   }
6   byteStream.write(buff);
7 }
```

Although obvious to a human that each line has a line number with it, a regular expression based solution will not suffice. The Java compiler automatically removes all unnecessary white spaces, so a line that begins with a number might be a perfectly valid line that conforms to the language.

```
result = 10 +
7 + 20; //A valid Java line
```

Suppression in Code This sub-category lies vis-a-vis the previous sub-category. Developers don't just add constructs, but also use omissions to help discuss code effectively. The following are the concepts belonging to this sub-category.

Missing Definitions and Missing Blocks (MD/MB) The snippet shown below, from this SO post [152] provides an example for the the case of suppression of structure. The Developer here chooses to present code from two separate compilable units in a single snippet.

```
public Foo{

}
package com.company.application;
public Bar extends Foo{
    private String appName = "MyFirstApp";
}
```

As can be seen from the snippet below, extracted from this SO post [153], suppression of structure can also occur at the intra-compilation unit level.

```
import com.google.gwt.user.client.Cookies;
Cookies.setCookie(name, value);
Cookies.getCookie(name);
```

The missing class instantiation here leads to fatal parsing issues. Entire class definition is irrelevant to this discussion. In another post [154] titled "What is the meaning of this line?", a developer asks “Can you explain these lines”, repeats the following line to gain focus and also gives the full code later.

```
public static final int GAMEPAD_UP = 0x0040;
```

Ellipses An Ellipsis is a punctuation used to indicate omission. As can be seen in the snippet from a real SO post [155], ellipses are also common in snippets. They are used as a place-holder for code that is irrelevant to the discussion.

```
public class DataForward extends Service{
private Context con = getBaseContext();
private Timer timer = new Timer();
<...>
@Override
public void onStart(Intent intent, int startId) {
```

The omitted code may be hard to guess but are irrelevant for the current discussion context. Following [156] is another example of ellipses:

```
Object[] params = ....;
String s = String.format("%S has %.2f euros", params);
```

But, not all instances of “...” is a parsing issue. For instance, the following post [156] on variable arguments is valid in Java:

```
public void foo(String... args) {
}

String args[] = new String[10];
foo(args);
```

It is also possible that the ellipse occurred in a string literal which will also not cause parsing problems. Such instances make it difficult to use simple regular expressions to detect the existence of this issue.

Substitutions for Code This group of modifications are those that serve as substitutes for code snippets themselves. This automatically makes snippets belonging to this group erroneous, since the alternatives leave out the necessities of the language.

Pseudocode Pseudocode is a notation resembling a simplified programming language. In this SO post [157], it is used for a high-level description of code while abstracting the syntactical specifics.

```
Step 1: Initialize an empty string. (say str)
Step 2: Construct a new 'Finch' object.
Step 3: BEGIN LOOP
           Fetch 'FinchMenu' from 'Finch' object.
           assign 'FinchMenu' to 'str'
           IF 'FinchMenu' is "Back and forward"
               Call 'RunAccelerationTest' method with
           'str' as argument.
           END IF
       END LOOP
```

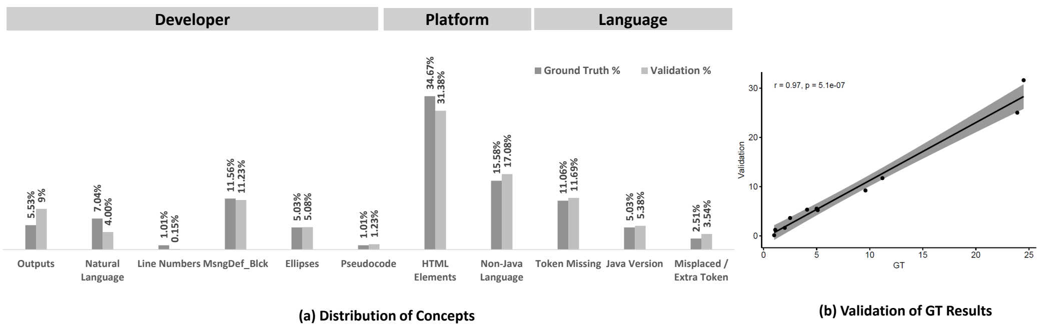


Figure 6.3: (a) Concepts and categories discovered using grounded theory approach. (b) When validated against a random sample of 500 snippets, we observe strong correlation.

6.4.2 Platform Related Issues

Some errors that make code snippets unclean are the result of the way the data is handled or presented by the platform, which in our case, is SO. SO dumps are XML files that contain information such as the text, code and ID of every post. Next, we discuss the Platform specific issues that cause unclean code.

HTML Issues The issue that arises with the code being embedded within HTML is that in some cases, the encoding might give rise to elements not native to the language. For example, $<$ and $>$ are encoded as &lt and &gt. This represents the need for a parsing stage that decodes such HTML encodings. The code snippet below is the result of extracting the content within the the `<code></code>` tags for this SO snippet [158] which also contains NL statements.

```
    int val = 1;
    for (int i = 1; i &lt;= n; i++) {
    //Note the use of &lt; above.
        val *= i;
        System.out.println(val); }
```

Non-Java Languages The snippet shown below is an XML snippet from an SO post [159] which has the “Java” tag. SO does not provide separate HTML markups to distinguish between multiple programming languages or even between actual code and any non-code constructs such as XML configuration files or JSON files. Hence, researchers should not assume that a snippet presented from a post tagged with the Java tag is indeed Java code.

```
<root>
    <Massage>No privillage</Mesaage>
    <result>
       <schma_index>
     <id>8</id>
     <name>raja</name>
       <schma_index>
    </result>
</root>
```

6.4.3 Language Related Issues

This category of errors are aggregate of those concepts that are due to syntactic issues.

Missing Tokens In the code snippet [160] shown below, the developer does not specify the tokens present within the `if` condition. As future work, we will focus on detecting and predicting the intent of the developer for code with such missing tokens.

```
for(int i = 0; i<theMessage.length;i+3){
    if ( ) //Missing conditional
    return ;
}
```

The above issue is a result of an intentional miss for the sake of emphasis or readability.

JDT Version Incompatibility of the JDT parser and the version of Java used in the snippet too give rise to errors. For example, for the snippet from post [161], JDT version 3.5.2 gives an error on try statement with parameters. But such a statement is accepted as a valid statement in the latest version of Java.

```
try
 (Socket clientSocket = new Socket(ipaddress, 7420)) {
       cp.updateGUI("Connection initiated...
                      waiting for outputs!"+"\n");
```

Misplaced/Extra Tokens Several errors during parsing arise due to misplaced or addition of irrelevant tokens in the code snippet. In the following example from an SO post [162], there is an additional assignment operator after the increment operator in the loop that results in an error during parsing.

```
final Map<Integer, List<String>> myMap =
       new HashMap<Integer, List<String>>();

  for (int i = 0; i < something; i++=) {
      myMap.put(i, new ArrayList<String>());
  }
```

6.5 Results

Figure 6.3 (a) shows us the distribution of each of the discovered concepts. We find that **50.25%** of the issues are caused by the Platform and **31.16%** of the issues caused by the Developer. Current works focus on issues such as namespace resolution and type inference. But Language specific issues constitute only **19.59%** of all issues. Furthermore, the issues introduced by Developers and the Platform are not addressed by such works and thereby make these approaches futile until such issues are resolved first. The aforementioned categorization of concepts is based on the cause of these errors. This serves to present the major causes, identify the dominant issues and assess the extent to which existing work would help solve the problem.

6.6 Building a Classifier to Validate Results

Our vision is to detect the issues identified in Section 6.4 while parsing source code snippets. We envision a classifier which can first categorize if the snippet is clean. If not, it outputs the concept that is most likely associated with the parsing issue in code.

We build a classifier which depends on the existence of one or more tokens to detect the presence of an issue. This intuition does not apply to all classes. For example, existence of a number does not mean that it is a line number. For some of the issue types, the issue is characterized by missing tokens. For example, `if () {}` misses tokens inside the `if` condition. Our classifier cannot detect these issues. For these kinds of issues where our classifier cannot validate, we perform additional manual validation. Hence, we limit the scope of our classifier to the following classes:

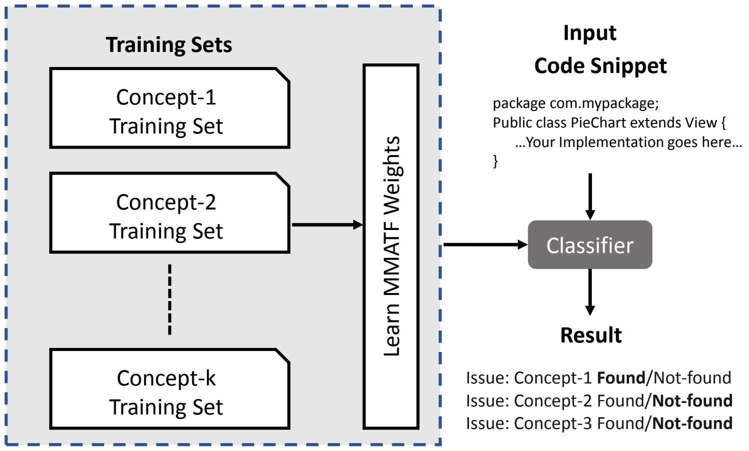
1. Outputs
2. Ellipses
3. HTML Elements

Ideally, we would like to cover all the listed issues. But, such a universal classifier will require significant research effort and might not be even possible to build at the first place. Hence, we have limited our attention to three classes. Our selection of classes is only based on the nature of the features describing the class being token-based.

We have adapted a state of the art Term Frequency model for computing term weights. Using training data gathered during the data analysis phase, we compute the term weights. The top-k terms by weight for each concept define the concept. For any input code snippet to the classifier, it computes the overall similarity score to the top-k terms of each concept. The jMechanic classifier declares the snippet as clean if the score obtained for every concept is below a threshold ψ. For those concepts where it finds the $score > \psi$, it picks the top ranking concept by score as the reason for the parsing issue. Figure 6.4 explains the workings of the classifier. Next we give the necessary terminology, background and explain our proposed Extended MATF model for Source Code (TF_{MX}).

We define any subsequence of tokens of a compilable program as a *Code Snippet*. Therefore, generally code snippets are not error-free compilable units. For example, method definitions or just a sequence of statements are typical code snippets shared on SO. The tokens can be keywords (ζ_{key}) or user-defined (ζ_{udt}). Examples of ζ_{key} are tokens corresponding to the Java keywords representing loops and branches. Tokens in the variable names, method names and comments are examples of ζ_{udt}. A code snippet $s = (e_1, e_2, ..., e_n)$ is a sequence of n tokens where $e_i \in \zeta_{udt} \cup \zeta_{udt}$. Our goal is to associate a set of issues $I \subset i_1, i_2, ..., i_7$ to each of the code snippets e found in SO. Here, i_j refers the parsing issues that were identified in our manual study.

We model this problem as a search problem for token sets that characterize each of these problems on a large collection of code snippets. For each concept identified in GT, we construct a goldset which is a mix of clean and unclean (of that specific category) code snippets from SO posts. We maintain the same split as shown in our GT results 6.3 to make the data representative of SO. We are inspired by the existing literature that leverage term



```
package com.mypackage;
Public class PieChart extends View {
    ...Your Implementation goes here...
}
```

Figure 6.4: jMechanic parse issue detection uses a classifier which learns from Mixed MATF weights.

saliancy [163, 9, 22, 24] to identify relevance of documents to a query. Since code snippets in SO have a wide distribution over length and distinct terms, we adapt the Multi Aspect Term Frequency (MATF) [38] model (see Section 2.1.2) to determine term salience.

Extended MATF Model for Source Code (TF_{MX}) In this work we deal with only source code snippets as content. However, each snippets may contain two token types, ζ_{key} and ζ_{udt}. If MATF is applied on this content, all token types get the same weight. We are interested in matching with user defined tokens more than keywords. Hence, we distinguish these two as separate components. Thus we use an extended multi-component adaptation of MATF. In the scenario that one component does not have any of the query terms, we do not want the overall MATF score to be zero. To avoid this scenario, we introduce a smoothing parameter α. Equation 6.1 gives this formulation. We have used the ideas of add-one or Laplace smoothing but with a much smaller value of $\alpha = 0.01$. These components are mixed in code. Hence, we propose an extended MATF (TF_{MX}) formulation as follows:

$$TF_{MX}(d,q) = \left(\prod_{c=1}^{n} [\gamma_c tf_{M_c}(t,p_c) + \alpha] \right)^{\frac{1}{n}} \tag{6.1}$$

where MATF score $tf_{M_c}(t,p_c)$ for each component is given as:

$$tf_{M_c}(t,p_c) = w_c \frac{tf_{R_c}(t,p_c)}{1 + tf_{R_c}(t,p_c)} + (1 - w_c)\frac{tf_{L_c}(t,p_c)}{1 + tf_{L_c}(t,p_c)} \tag{6.2}$$

In our case, the components in code are (c = 1) keywords and (c = 2) user defined items.

The MATF score for each component is computed using the tf_{M_c} formulation in Equation 2.4. The parameter p_c denotes a specific component in the post such as title. We use the notation tf_{R_c}, tf_{L_c} and w_c to denote component-wise scores for RITF, LRTF and aspect weight respectively. We use the same $\frac{2}{1+log_2(1+|Q|)}$ formulation for the aspect weight w_c. The $MATF_{src}$score is then a geometric mean over the weighted MATF scores of each component. $use(t, c)$ is a boolean function which returns one if the term t should be used in the computation for the component c, else zero.

Component Weight (γ_c) Different components carry different amount of term saliency in the corpus. For instance, in the case of SO posts, developers ensure that the same question is not asked before by searching through several titles of existing posts. Therefore, they carefully choose relevant terms for the title. Hence, title terms should carry a higher weight compared to the rest of the components in SO. We use γ_c to refer to the weight of each component.

Term Frequency (tf_{L_c}, tf_{R_c}) The LRTF (tf_{L_c}) and RITF (tf_{R_c}) values correspond to the length normalized and simple counts of terms in the document, respectively. This works for text. However, for source code, we need to do additional processing to extract the tokens of interest. For source code, we extract the AST and from there, we get the identifier tokens. Each identifier is processed for stemming and stopwords. The resulting tokens of code constitute the terms of the component.

We apply the standard IDF measure. We compute the similarity between query and post vectors, as follows:

$$SIM(Q,p) = \sum_{i=1}^{|Q|} tf_{MC}(q_i, p) \times \log_2\left(\frac{N}{df(q_i)}\right) \tag{6.3}$$

A reasonable TF-IDF model should satisfy Fang's constraints [90]. Next, we validate TF_{MX} against these constraints. TF constraint states that "*Assume* $|d_1| = |d_2|$. *If* $c(w, d_1) > c(w, d_2)$, *then* $tf(d_1, q) > tf(d_2, q)$". Here, $c(w, d_1)$ refers to the count of word w in document d_i. While proposing this constraint, they assume that the documents are homogeneous. We extend this constraint to mixed component corpora as follows:

- *TF_{MX} Constraint 1*: Let $q = w$ be a query with only one term w. Assume $\forall_{c=1}^{n}(\tau(c_i, d_1) = \tau(c_i, d_2))$. If $\forall_{c=1}^{n}(\tau(w, c_i, d_1) > \tau(w, c_i, d_2))$, then $tf(d_1, q) > tf(d_2, q)$. Here, $\tau(w, c_i, d_j)$ refers to the count of word w in component c_i of document d_j. $\tau(c_i, d_j)$ refers to the size of the component c_i in document d_j.

TF_{MX}satisfies this modified constraint. Similarly, the following constraint holds for TF_{MX}:

- *TF_{MX} Constraint 2*: This constraint ensures that tf increase is lesser for larger TF values when the word count increase remains the same. For example, a word count increase from 1 to 2 contributes more to tf than an increase from 100 to 101. Let $q = w$ be a query with a single term w. Assume that $|d_1| = |d_2| = |d_3|$ and $\forall_{c=1}^{n}\ \tau(w, c_i, d_1) > 0$. If $\forall_{c=1}^{n}(\tau(w, c, d_2) - \tau(w, c, d_1) = 1)$ and $\forall_{c=1}^{n}(\tau(w, c, d_3) - \tau(w, c, d_2) = 1)$, then $tf(d_2, q) - tf(d_1, q) > tf(d_3, q) - tf(d_2, q)$.

Fang et al. [90], also propose a length normalization constraint and a IDF constraint. Since we use the standard IDF measure and MATF length normalization, those constraints do not change.

6.7 Implementation and Evaluation of Classifier

We have implemented TF_{MX} as a standalone reusable component. We have also implemented and shared jMechanic tool which uses TF_{MX} to identify parsing issues in the input code snippet. Although jMechanic itself has been implemented in Java and is used to detect parsing issues in Java snippets, nothing in the approach or implementation makes it Java specific. A training set of clean and unclean snippets of a specific issue is sufficient for jMechanic to statistically learn about the characteristics of the parsing issue.

Goldset Preparation To evaluate the classifier, we construct a goldset. We prepare a goldset containing clean snippets manually mixed with unclean snippets for each concept identified from our study. The mix contains 10 snippets suffering from a specific issue in the same proportion identified by our study. For example, our GT study reports that ellipses form 5% of all erroneous snippets. So our goldset contains 10 (5%) snippets which suffer from ellipses and 190 (95%) clean snippets. This is done to ensure that the index constructed using TF_{MX} model are representative of the composition of issues in entire SO.

Index Construction and Expert Curation For each distinct term in the unclean snippets of the goldset thus created, we compute the TF_{MX}-IDF score. We extract the top 15 terms ordered by this score as *descriptive terms* of the concept in unclean snippets. An expert then looks at the descriptive terms and decides on the relevance of each term. The expert may add or remove terms to arrive at the final list of descriptive terms. We then proceed to build a classifier with these *descriptive terms* as *features*.

Classifier Design Given a code snippet, our classifier tokenizes the input code snippet and for each known concept and checks whether any of the tokens are *descriptive terms* for the given issue. The total score (S) of the snippet is the length normalized sum of the weights (TF_{MX}-IDF) of each of the descriptive terms multiplied by its frequency (N) in the given snippet as shown in Equation 6.4.

$$S = \frac{1}{n}\sum_{i=1}^{n} tf_{MX}.IDF \tag{6.4}$$

Classifier declares the snippets with score greater than threshold ($S > \psi$) as to suffer from the given issue. We compute this threshold empirically.

To do so, we prepare a testset similar to the goldset with a mix of clean snippets and erroneous snippets suffering from the issue in the ratio of 1:1. We use it to extract the descriptive terms for different thresholds. We also compute the F_1 score of our classifier for each threshold. We find the best threshold per concept that maximizes the F_1 score. Thus, we empirically derive the threshold values.

Since each snippet can suffer from multiple issues each of which is independent of the other, our classification task becomes a multi-label-binary classification problem. Hence our classifier functions as a set of binary classifiers, one for every kind of issue.

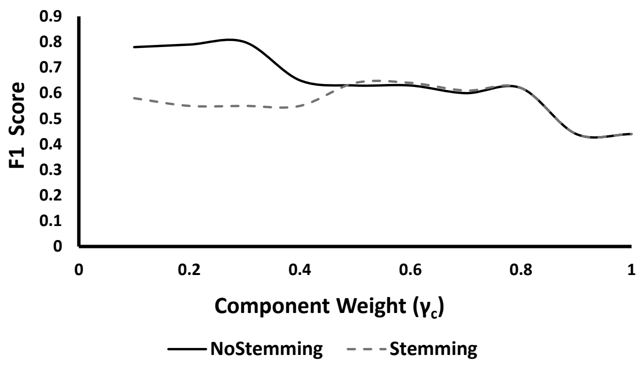


Figure 6.5: A component weight setting of 0.2 works best for our work.

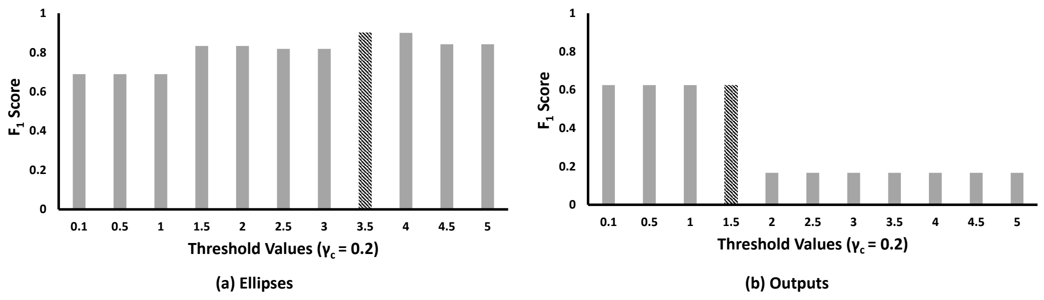


Figure 6.6: Examples of Threshold Parameter Estimation

6.7.1 Parameter Tuning

We derive the component weight (γ_{key}) and the thresholds empirically. Figure 6.5 shows that for $\gamma_{key} = 0.2$, we get the highest average F_1 scores across all issues. One explanation for this observation is that non-key terms such as "&lt;" and "&gt;" are usually the reason for parsing issues and hence relatively high thresholds (downscaling the importance of key terms) for non-key terms gives better TF_{MX} values.

To empirically identify the optimal issue specific thresholds, we pick the threshold corresponding to the optimal F_1 scores of a given issue. Figure 6.6 (a) and Figure 6.6 (b) shows the change in average F_1 scores as we change the threshold values for the ellipses and pseudocode respectively.

Table 6.1: Precision, Recall and F_1 Score of the classifier. Our classifier gives good results for all cases where term saliency can be observed.

Class	**Precision**	**Recall**	F_1 **Score**
Ellipses	0.90	0.90	0.90
Output	1.00	0.46	0.63
HTML Elements	0.73	1.00	0.84

Table 6.2: Threshold Values and Descriptive Terms extracted using the *Extended MATF Model for Source Code.*

Class	**Threshold (ψ)**	**Top-3 Descriptive Terms**
Ellipses	3.3	... , args , string
Output	2.1	thread , "main" , unknown
HTML Elements	1.4	&lt;, &amp;

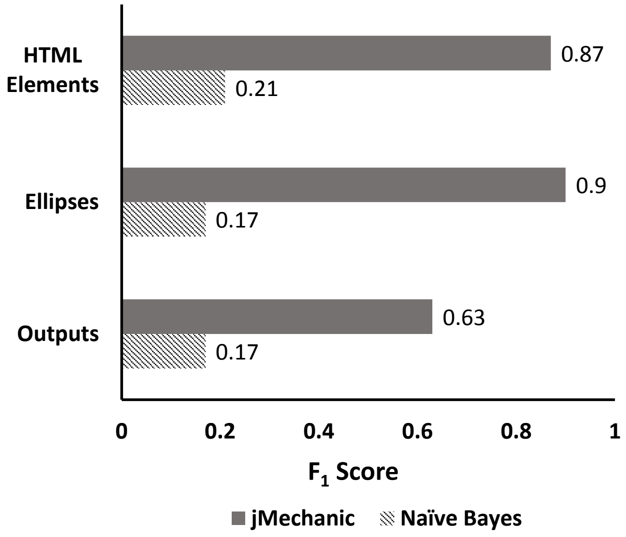


Figure 6.7: We compare the results of manual study on limited snippets with the results of the classifier on entire SO. We find that the correlation is significant for all the classes. Comparison with Naive Bayes shows that our classifier performs 45% better.

6.7.2 Comparative Evaluation

To test the efficiency of our classifier, we compare our classifier with a Naive Bayes Classifier built using Weka[3]. We use term frequency of all distinct terms as the features provided to the Naive Bayes classifier. Naive Bayes based classifiers are simple probabilistic classifiers. They assume independence between terms. In spite of their simplicity, they are shown to be effective in several studies [164, 165]. Hence, we use it as a benchmark to evaluate our classifier performance. On the same set of clean and erroneous examples that we use to train our classifier, we train Naive Bayes classifier as well. Figure 6.7 shows the results of this comparative evaluation. We find that our classifier improves the F_1 scores on average by 30% compared to Naive Bayes. Moreover, our classifier outperforms ZeroR by 38% on F_1 score.

[3]https://www.cs.waikato.ac.nz/ml/weka/

Table 6.3: Statistics on the parseable and erroneous snippets in Java tagged posts in Stack Overflow.

	Total	**Erroneous**	**Clean**	**Non-Java**
Java tagged snippets in SO	1.47 M (100%)	1.01 M (68.4%)	0.46 M (31.6%)	Not Computed
After curation with jMechanic	1.47 M (100%)	0.52 M (35.1%)	0.93 M (63.3%)	0.02 M (1.6%)

Table 6.4: The percentages represent the fraction of erroneous snippets that suffer from the given issue. GT Study represents numbers from our initial study. GT Validation shows the results from manual validation. Last column gives the results observed after running the classifier on entire dump.

Concept	**GT Study**	**GT Validation**	**Classifier-based Validation**
Ellipses	5.03	5.08	9.12
HTML	34.67	31.38	34.90
Outputs	5.53	9.23	5.85

6.8 Validating the Study Results using the Classifier

We find that **50.25%** of the total issues are platform related issues and hence they dominate the concepts. Developer issues are the next most significant category. Among, developer issues, *Missing Definitions and Missing Blocks* forms the major concept covering **11.56%** of the issues. While most tools focus on type inferences and such language issues, the category including all three concepts constitute to only **19.59%** of the total issues. Figure 6.3 (a) gives the entire distribution. This validates the need for us to focus on Developer and Platform categories for tool building. In this preliminary work, we fix the issues marked in Figure 6.2 in a tool named jMechanic. jMechanic uses string manipulation and heuristics to reduce 3 out of the 11 issues. On running jMechanic on the entire 12/2017 SO dump[4] containing 1.47 Million snippets, jMechanic is able to curate snippets so that **63.3%** of the input snippets produce an AST. Without jMechanic, only **31.6%** snippets were clean (Table 6.3).

6.8.1 Validation

Manual Validation To validate our theory, we randomly sampled 500 snippets and manually categorized them into the identified buckets. Figure 6.3 (b) shows the concepts and the number of snippets from the original set used for arriving at the concepts and the corresponding number from the validation set. There exists a linear correlation between them. The correlation is strong and statistically significant with Spearman co-efficient $\rho = 0.98$ at $p = 8.6e - 08$.

[4]https://archive.org/details/stackexchange

Validating using the Classifier Manual validation was limited to 500 snippets. To reduce the threat that another 500 snippets may give different results, we validate our theory using our classifier on all snippets in SO. Once we train it over all the concepts, we run it across all erroneous snippets in the SO dump with the empirically found thresholds to validate the results. As shown in Table 6.4, the results from our study, validation and the classifier results are off by only less than 5%. This validates our theory on the dominating issues, for the three concepts.

6.8.2 Effectiveness of jMechanic

To know if we have captured all the dominant issues, we conducted a study over 3 researchers. These researchers had at least two years of experience in Java coding and had authored at least one research paper involving big data. We selected 10 unclean snippets randomly from the goldset and showed them one by one to each of the participants. We asked them to select the most appropriate parsing issues associated with each snippet. We listed all 11 concepts and added an "Other. Please specify." option. No participant selected the "Other. Please specify." option. We compare the user given answers with jMechanic response. At top-1 and top-3 answers, our classifier gives a precision of 80% and recall of 75% when the user given answers were considered as ground truth. We had an inter-rator agreement of 100%. Note that we did not capture or compare the ranking of the issues as that is not our focus. Finally, we asked if the participants like a tool to automatically identify parsing issues, and extract clean snippets from SO. All the participants agreed that such a tool will be useful.

There are several cases where jMechanic failed. For instance, jMechanic could not differentiate between the ellipses ("...") seen in variable arguments or varargs based snippets as in `f(Object... args)` and the ellipses that developers leave to abstract the code. Several snippets marked as code could not be identified as Outputs especially when the output contained API references or similar terms. In some cases, line numbers were wrongly identified as output. Our focus here is to show that a knowledge discovery approach can be used to solve this problem. Our approach gives promising results to explore further in this direction.

6.9 Related Work

Characterization Studies on Code Snippets in SO Our work makes the novel contribution of characterizing unclean code snippets. Prior characterization works such as [166, 167, 168, 169], based on Q&A websites have been predominantly about the quality of posts. These studies aim to predict or identify good or bad posts. Works such as [170, 171, 139] address the problem of the quality of code snippets. Nasehi et al. [170] determine what a good code example is using a score computed from the number of upvotes and downvotes an answer has received. Tavaloki et al. [171] tackle the problem of infeasibility of using code snippets directly from the web. However, they do not define quality of snippets from the perspective of parsing. The goal of the study by Yang et al. [139] is to compare the usability rates for snippets of four programming languages. To the best of our knowledge, our work is the first to characterize the reasons behind code snippets being unclean.

Code Extraction and Snippet Curation One of the largely ignored problems that exists in existing works is the separation of code from the non-code parts of a post. Ponzanelli et al. [172] build a H-AST that represents all the components of a post in the form of a tree-like data structure. Their work is based on an earlier work by Bacchelli et al. [173] who present the idea of using Island Grammars to separate structured text from unstructured NL text. Curation of code snippets is addressed by several researchers [142, 171, 30, 143] but without characterizing at the first place. This makes these tools unsuitable for large data dumps. The works of Zhong et al. [143] and Dagenais et al. [30] address problems specific to the *Language* category which accounts for the least of all the categories.

Classification of Code Snippets Zhang et al., [164] classify code snippets for defect prediction. Xie and Engler [174] show that redundancy in source code lead to defects in code. Brun and Ernst [175] make a case that program properties exist which reveal errors. They use these properties to build a classifier to identify faulty programs without the help of test suites. Wang et al., [176] show that program semantics can be captured into defect prediction features. They too build a classifier for defect prediction. Yet, none of these efforts are focused towards parsing problems. We have implemented a classifier to identify parsing issues in code snippets.

6.10 Threats to Validity

We have studied Java code snippets only. Our findings may not generalize to other programming languages. However, nothing in the approach or the classifier design is tied to Java. Even the parsing issues such as “Natural Language” and “HTML elements” occur in code snippets belonging to other programming languages as well.

We have worked with code snippets found in SO. We may get different results on another discussion forum. To reduce this risk, we have supplemented our manual validation using 500 random snippets with a classifier thereby validating against the code snippets in entire SO.

Our results may not generalize beyond the manually sampled and verified list of 500 posts. To minimize this risk, we built and used a classifier to validate the results over the entire SO. The validity of the parsing issues distribution depends on the accuracy of the classifier. We have shown that the classifier is has an average F_1 score of 0.8 over 3 concepts.

6.11 Summary

Parsing errors in partial programs show up in discussion forums due to three major reasons: 1) Developer behavior, 2) Syntactical issues existing in the program before they get into the discussion forums, and 3) Platform related issues. We identify 11 major causes that together contribute to these categories. We find that the top-3 issues leading to parsing problems are 1) HTML Issues 2) Natural Language in Code 3) Non-Java Code. Different markups to embed non-Java items will be useful for researchers to extract and use code. Code extractors should focus on these issues to maximize the utility of SO. We have implemented part of these findings in a tool named jMechanic. It is able to extract AST out of 63.3% of the 1.47 Million snippets in SO. We plan to improve jMechanic to parse more snippets by curating all the issue categories. We would also like to try jMechanic on other programming languages, especially of the non-imperative kind.

Chapter 7

Conclusion and Future Work

We show that code variants form an important class of code snippets. Variants are different from known snippet types such as simions and clones. We do not only propose an unambiguous definition of code variants but also characterize it. Our variant characterization presents several opportunities and challenges for tool support and automation. To automate the search for variants, we propose jSense structural and retrieval models. Our experiments on developers' perspective of code similarity lead us to a structural model of source code. Our experiments suggest that existing retrieval models do not work well with source code. Towards this end, we propose a Multi-Component Multi Aspect Term Frequency based retrieval model which we call the jSense Retrieval Model. We put these ideas to mine code variants in a tool named jSense. These models and methods enable searching over partial programs of discussion forums with a precision of 92% and recall of 71%. With ever growing volume of discussion in these forums, the approach will become more effective in recognizing topics and mining code snippets that serve as implementation choices. We observe that natural language queries on source code do not fare well due to the mismatch in representation of query terms and terms in source code. To solve this problem, we visualize source code as a collection of entities. We discover the entities and their surface forms in source code. We use this knowledge to improve code search. Our user studies show that this approach allows them to search faster by 29%. Finally, as we are limited by the number of snippets that we can parse from the web, we investigate the extent to which parsing problems exist. In SO, we show that only 31.3% of the code snippets parse to produce an Abstract Syntax Tree. Our grounded theory study of the issues suggests that there are primarily three kinds of issues which are due to developers, platform and language respectively. With the insights from this study, we are able to increase the snippets that parse, to 61%. With the ever increasing open source code, developer discussions and bug repository content, our techniques will do even better. This opens up all new field of research to investigate how to leverage the knowledge around by using information retrieval techniques.

7.1 Resources

The project resources such as datasets, source code and related documentation can be accessed from the project website[1].

7.2 Future Work

The jSense structural and retrieval models need to be extended for better precision and recall. Our work is limited to Java snippets. We would like to experiment with non-Java languages and languages of non-imperative kind. Apart from these, we also consider the following future work.

7.2.1 Applications of Code Variants

We have shown a method to construct a knowledge base of code variants. Such a knowledge base can be used to solve a wide variety of problems. Here, we list a few that form our future plans.

7.2.1.1 Propagating Assignment Feedback

In a MOOC setting, where instructor's time is limited and a large number of assignments need to be assessed, distinct variants of the assignments can be selected for manual feedback. This feedback can then be percolated to rest of the assignments where the same feedback is applicable.

7.2.1.2 Plagiarism Detection

In an academic setting, a variant knowledge base can be leveraged for plagiarism detection. Students who have used distinct techniques can be separated into separate clusters. This could reduce the load on plagiarism detectors.

7.2.1.3 Detection of Semantic Clones

As discussed in Section 3.2, there is no consistent definition of semantic clones. However, a knowledge base of variants can help semantic clone detectors to improve their precision. They can now skip those code snippets which they earlier suspected to be a semantic clone and refactored.

7.2.1.4 Summary

The ability to generate a knowledge base of code variants helps us to attack a wide variety of problems that are otherwise difficult to solve using traditional program analysis techniques.

[1]http://vvtesh.co.in

7.2.2 Modeling Tasks as Search Problems over Source Code

Many software engineering and other problems can be modeled as search problems over source code. Arnaoudova et al. [25] claim that more than 20 software engineering tasks are addressed through IR and NLP methods. Here, we discuss some of our related ongoing work and future plans.

7.2.2.1 Automated Programming Quizzes

As an application of our ability to search in SO which contains multiple components including text and code, we are working on arriving at automated programming quizzes. We can model this as a search problem over SO using MC-MATF. Majority of existing quizzes on programming languages focus on multiple choice or short answer questions like "*What is the output of the program?*" instead of questions like "*When is it better to use a 'while' loop instead of a 'for' loop?*". An initial prototype using an entity driven approach as in ANNE gives promising results [177]. Entities having similar attributes are exploited to discover patterns. For example, developers "declare" and "add elements" to *collections*. Since "Set" is a *collections*, we can infer that similar actions happen over them as well. Thus, Jain [177] shows that a knowledge discovery approach can be devised to arrive at useful questions.

7.2.2.2 Bug Detection

Many defects are caused by minor deviations from their respective clean implementation. Some examples are missing guard statements, improper looping, or a relaxed conditional. Using a repository of correct example code snippets, we can locate the defects in any given code snippet and suggest corrections. Similar ideas have been used in plagiarism detection [32] and programming assignment grading [31]. We take this idea to web-scale and look to provide local (within-IDE) recommendations for correction. To accomplish the above, we assume the availability of labeled correct solutions. Multiple such occurrences are required to provide precise mapping of the code-contexts. We call this the dense code assumption. In a typical classroom assignment, solutions for the same problem is submitted by many students which creates a dense code situation. Similarly, on web-scale, same ideas are discussed at multiple places in discussion forums. These discussions can be clustered to build a repository of correct examples for concepts such as factorial. Once we have such a repository, structural and semantic matches can be performed to find similarity of given code against the repository. Minor deviations are captured and reported as bugs.

7.2.2.3 Summary

Modeling software engineering tasks as search problems over source code provides promising results. Yet, the tools and techniques for code search is still not as mature as text search. We are hopeful that our results with MC-MATF and Selective Set Structure Indexing Methods will help us solve more problems of similar kind.

7.2.3 Searching in Software Binaries

So far, we have discussed code search. Binaries too make an interesting case for search. Specifically, we envision a tool to search for Mathematical Expressions (**ME**) in software binaries. A text search for "C++ Math Library" in SO[2] results in 4K posts. Developers often search for libraries that implement a certain mathematical expression. For example, a developer asks for a C library that implements Fast Fourier Transform in Quora[3]. We find 39K GitHub[4] projects which use the term "math" in their documentation. According to a study by Zhao et al. [178], users often look for resources such as code or a toolkit with an implementation of a **ME**. Hence, a search system for **ME** in binaries will be useful to developers. Such a search system can be used not only at development time for code reuse, but also be used by other stakeholders such as security analysts to locate vulnerabilities and software testers for bug detection. Search in binaries pose a very different set of challenges and opportunities.

7.2.3.1 Challenges and Opportunities in Dealing with Binaries

Variants Programs compiled using different compilers or with different optimization levels may result in dissimilar binaries. A program implementing x^2 when compiled without optimization, calls the *pow* function, with binary signature <*pow@plt*>. Whereas, an optimized version uses *mulsd* to multiply the number to itself. The instructions may differ for different compiler optimization levels such as O0, O2 and Os (provided by GNU compiler collection and implemented in gcc and g++).

Ghost Ops Not all instances of arithmetic opcodes in a binary provide insight about **ME**. Common actions such as passing arguments to a function on the stack and allocating memory make use of arithmetic opcodes too. We call such arithmetic opcodes, that do not have an explicit equivalent operator in source code, *Ghost Ops*. Hence isolating arithmetic opcodes with an equivalent operator in a mathematical expression is a challenge. For example, the presence of the *sub* instruction at assembly level need not imply that there exists a subtraction operation in the source code.

Evaluation Ordering Compilation may result in a binary where strands of the implemented expression may appear in any order. For example, a compiler may evaluate $a*b+c/d$ as $a*b$ followed by c/d or c/d followed by $a*b$ before finally performing the addition operation. Due to this, the order of operations in **ME** differ from those that surface in the binaries. Hence, to compute similarity, a specific sequence of operations cannot be assumed. We refer to this challenge as *Evaluation Ordering*.

Operand Resolution Since all operations at the assembly level are performed on registers or on values in memory, resolving the operands to variables is not a straightforward task. We call this challenge *Operand Resolution*. The resolution of operands plays a major role

[2][Oct 2017] https://stackoverflow.com/

[3]https://www.quora.com/Are-there-any-libraries-in-C-to-implement-FFTs

[4]https://github.com/

because if expressions were to be compared purely based on structure, the expression $b^2 - 4ac$ would be equal to $b^2 - 4ab$.

7.2.3.2 Challenges and Opportunities in Dealing with Mathematical Expressions

On a similar note as dealing with binaries, dealing with **ME** is also challenging for the following reasons:

Specification of Expressions Content MathML [179] (henceforth referred to as ContentML) provides a standardized way to capture **ME**. Yet, mathematical operators may have distinct forms. For example, $x * y$, $x \times y$ and xy represent the same expression. ContentML normalizes the representations, removing ambiguity.

Types of Operations In this work, we focus on algebraic and transcendental expressions. Algebraic expressions are those which can be represented using only algebraic operations, which consists of addition, subtraction, multiplication and division. Transcendental expressions by contrast, are those expressions that cannot be represented by a finite sequence of algebraic operations. The operations which make up transcendental expressions include exponential, logarithm and trigonometric functions. This is a challenge because we need to consider all the diverse ways of representing these operations and functions at the binary level. For example, in an unoptimized version *log* is represented by <*log*@*plt*>, while in an optimized version it may get replaced by some precomputed value.

A more detailed discussion of this work is presented in Ridhi et al.'s [180] paper. Going forward, we look at developing an end-to-end search system to address these challenges.

7.2.3.3 Summary

We envision automating the creation of a KB for multiple system architectures. Apart from the classes of operations considered, there are other classes of operations such as logical (&& for AND, $\rightarrow$ for implies), and relational (such as $\leq$). Summation ($\sum$) and product ($\prod$) are examples of *iterative operations* that require applying an expression over a range of values. Precision can be improved by keeping track of the operands in the **ME**. We will address these in our future work. Our work opens up a wide range of opportunities to attack problems on searching domain specific (such as music, medical and finance) content in binaries. We find that knowledge base assisted solution is promising to address such problems.

Appendix A

User Study - Searching for Variants

Developers/ Question	**Dev1**	**Dev2**	**Dev3**	**Dev4**	**Dev5**	**Dev6**	**Dev7**	**Dev8**	**Dev9**
Would you have coded a factorial like that in Listing 3?	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Time taken to review the code snippet	4	6	2	3	5	3	3	2	3
What,could be wrong with this example? Can you identify some potentially missing parts in computing factorial, or some related bugs?	Debug statements are missing	No issues	Handle exceptions, n can be large.	Better naming conventions	Better comments and exceptions	Conventions might be violated. Functionally looks ok.	I have seen hardcoded implementations. This calculates everytime.	We can cache results as and when we do this computation once.	Looks good.
After seeing variants: (will you still write it this way?)	Not Sure	No.	Not Sure.	Yes	Don't know.	No	No	Depends on the requirement.	May be. Depends on requirement.
What changes will you consider making to this snippet?	0! and negative input...[1]	it needs some non-trivial extensions...[2]	n cannot be int...[3]	Unless the context demands...[4]	I don't know...[5]	BigInt, negative n.	Large n, negative n, zero n, checks.	Hardcode results. Cache values. Large n.	Exceptions, Input validations, comments, API usage.
Do you think developers will benefit from looking at variant implementations?	Yes	Yes	Less useful for short examples. Large examples won't have variants available for reference.	Yes...[6]	Yes	Yes	Yes	Yes	Yes. Helps to learn what could potentially go wrong.

[1] 0! and negative input should be explicitly handled. int as type might be wrong.

[2] it needs some non-trivial extensions. Input validation (0!), Use BigInt, Handle Exceptions, Hardcode Results.

[3] n cannot be int, negative numbers as input should be validated, Recursion looks more elegant.

[4] Unless the context demands I will still keep it simple. There may be possible extensions but I want to see why we need them from either requirements or program context. If nothing more is mentioned, I will keep it as it is. Writing "fast factorial" is a new requirement. I don't see any difference between recursive and iterative versions in the context of professional code.

[5] I don't know. I will start with the same version and perhaps extend it later if required.

[6] Yes. I would certainly like to learn at least and know what could potentially go wrong.

Bibliography

[1] Jonathan Sillito, Frank Maurer, Seyed Mehdi Nasehi, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 25–34, Washington, DC, USA, 2012. IEEE Computer Society.

[2] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the search for source code. *ACM Trans. Softw. Eng. Methodol.*, 23(3):26:1–26:45, June 2014.

[3] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. ASE '01, pages 107–, Washington, DC, USA, 2001. IEEE Computer Society.

[5] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. ICSE '08, pages 321–330, New York, NY, USA, 2008.

[6] J. Singer and T. Lethbridge. What's so great about 'grep'? implications for program comprehension tools. In *Technical Report, National Research Council, Canada.*, 1997.

[7] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Eng. Methodol.*, 21(1):4:1–4:25, December 2011.

[8] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: A case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 191–201, New York, NY, USA, 2015. ACM.

[9] Renuka Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 905–908, New York, NY, USA, 2006. ACM.

[10] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 234–243, New York, NY, USA, 2007.

[11] Bunyamin Sisman and Avinash C. Kak. Assisting code search with automatic query reformulation for bug localization. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 309–318, Piscataway, NJ, USA, 2013. IEEE Press.

[12] krugle. `http://opensearch.krugle.org`. Last accesed: 23-Jan-2018, 2018.

[13] searchcode. `https://searchcode.com`. Last accesed: 23-Jan-2018, 2018.

[14] Meili Lu, X. Sun, S. Wang, D. Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 545–549, March 2015.

[15] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. Rack: Code search in the ide using crowdsourced knowledge. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 51–54, Piscataway, NJ, USA, 2017. IEEE Press.

[16] Raphael Sirres, Tegawendé F. Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. Augmenting and structuring user queries to support efficient free-form code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 945–945, New York, NY, USA, 2018. ACM.

[17] Shaowei Wang, David Lo, and Lingxiao Jiang. Autoquery: Automatic construction of dependency queries for code search. *Automated Software Engg.*, 23(3):393–425, September 2016.

[18] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 457–466, New York, NY, USA, 2010. ACM.

[19] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina 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.

[20] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina 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.

[21] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le 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.

[22] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015.

[23] Pliny. `http://pliny.rice.edu/`.

[24] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 102–111, New York, NY, USA, 2014.

[25] Venera Arnaoudova, Sonia Haiduc, Andrian Marcus, and Giuliano Antoniol. The use of text retrieval and natural language processing in software engineering. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 949–950, Piscataway, NJ, USA, 2015. IEEE Press.

[26] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Grapacc: A graph-based pattern-oriented, context-sensitive code completion tool. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1407–1410, Piscataway, NJ, USA, 2012. IEEE Press.

[27] DARPA. Muse envisions mining "big code to improve software reliability and construction, 2014. `http://www.darpa.mil/news-events/2014-03-06a`.

[28] G. W. Furnas, S. Deerwester, S. T. Dumais, T. K. Landauer, R. A. Harshman, L. A. Streeter, and K. E. Lochbaum. Information retrieval using a singular value decomposition model of latent semantic structure. In *Proceedings of the 11th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '88, pages 465–480, New York, NY, USA, 1988.

[29] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 157–166, New York, NY, USA, 2010.

[30] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 313–328, New York, NY, USA, 2008.

[31] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 491–502, New York, NY, USA, 2014.

[32] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003.

[33] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[34] Giacomo Domeniconi, Gianluca Moro, Roberto Pasolini, and Claudio Sartori. A study on term weighting for text categorization: A novel supervised variant of tf.idf. In *Proceedings of 4th International Conference on Data Management Technologies and Applications*, DATA 2015, pages 26–37, Portugal, 2015. SCITEPRESS - Science and Technology Publications, Lda.

[35] Amit Singhal, Chris Buckley, and Mandar Mitra. Pivoted document length normalization. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '96, pages 21–29, New York, NY, USA, 1996. ACM.

[36] Hui Fang, Tao Tao, and Chengxiang Zhai. Diagnostic evaluation of information retrieval models. *ACM Trans. Inf. Syst.*, 29(2):7:1–7:42, April 2011.

[37] Venkatesh Vinayakarao, Rahul Purandare, and Aditya V. Nori. Structurally heterogeneous source code examples from unstructured knowledge sources. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, pages 21–26, 2015.

[38] Jiaul H. Paik. A novel tf-idf weighting scheme for effective ranking. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 343–352, New York, NY, USA, 2013. ACM.

[39] Peilin Yang and Hui Fang. A reproducibility study of information retrieval models. In *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval*, ICTIR '16, pages 77–86, New York, NY, USA, 2016. ACM.

[40] J. Cordeiro, B. Antunes, and P. Gomes. Context-based recommendation to support problem solving in software development. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 85–89, June 2012.

[41] Zhiyuan Cai, Kaiqi Zhao, Kenny Q. Zhu, and Haixun Wang. Wikification via link co-occurrence. In *Proceedings of the 22Nd ACM International Conference on Conference on Information &#38; Knowledge Management*, CIKM '13, pages 1087–1096, New York, NY, USA, 2013.

[42] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar 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.

[43] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. Code similarities beyond copy & paste. CSMR '10, pages 78–87, Washington, DC, USA, 2010. IEEE Computer Society.

[44] Rainer Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006*, 2006.

[45] I. Keivanloo, C. K. Roy, and J. Rilling. Sebyte: A semantic clone detection tool for intermediate languages. ICPC '12, pages 247–249, June 2012.

[46] Rochelle Elva and Gary T. Leavens. JSCTracker: A semantic clone detection tool for Java code. Technical Report CS-TR-12-04, Computer Science, University of Central Florida, Orlando, Florida, March 2012.

[47] Alan J. Perlis and Spencer Rugaber. Programming with idioms in apl. *SIGAPL APL Quote Quad*, 9(4):232–235, May 1979.

[48] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. FSE 2014, pages 472–483, New York, NY, USA, 2014.

[49] Oleksandr Panchenko, Hasso Plattner, and Alexander Zeier. What do developers search for in source code and why. SUITE '11, pages 33–36, New York, NY, USA, 2011.

[50] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. OOPSLA '06, pages 413–430, New York, NY, USA, 2006.

[51] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? ICSE '15, pages 880–890, Piscataway, NJ, USA, 2015.

[52] Jonathan Sillito, Frank Maurer, Seyed Mehdi Nasehi, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 25–34, Washington, DC, USA, 2012. IEEE Computer Society.

[53] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM.

[54] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. Recommending source code examples via api call usages and documentation. RSSE '10, pages 21–25, New York, NY, USA, 2010.

[55] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. EuroSys '09, pages 33–46, New York, NY, USA, 2009.

[56] Saurav Muralidharan, Amit Roy, Mary Hall, Michael Garland, and Piyush Rai. Architecture-adaptive code variant tuning. ASPLOS '16, pages 325–338, New York, NY, USA, 2016.

[57] Julia Rubin and Marsha Chechik. A framework for managing cloned product variants. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1233–1236, Piscataway, NJ, USA, 2013.

[58] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu. How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 381–392, May 2017.

[59] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[60] Chanchal K. Roy and James R. Cordy. Are scripting languages really different? In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 17–24, New York, NY, USA, 2010. ACM.

[61] Wai Ting Cheung, Sukyoung Ryu, and Sunghun Kim. Development nature matters: An empirical study of code clones in javascript applications. *Empirical Software Engineering*, 21(2):517–564, 2016.

[62] Jens Krinke. Effects of context on program slicing. *J. Syst. Softw.*, 79(9):1249–1260, September 2006.

[63] Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. Softw. Eng.*, 26(1):15–35, January 2000.

[64] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. ICSE '12, pages 69–79, Piscataway, NJ, USA, 2012.

[65] Martin P. Robillard. Topology analysis of software dependencies. *ACM Trans. Softw. Eng. Methodol.*, 17(4):18:1–18:36, 2008.

[66] Robert P. Nix. Editing by example. *ACM Trans. Program. Lang. Syst.*, 7(4):600–621, October 1985.

[67] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, April 2012.

[68] Martin C. Rinard. Example-driven program synthesis for end-user programming: Technical perspective. *Commun. ACM*, 55(8):96–96, August 2012.

[69] Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. OOPSLA '14, pages 491–506, New York, NY, USA, 2014.

[70] Aritra Dhar, Rahul Purandare, Mohan Dhawan, and Suresh Rangaswamy. Clotho: Saving programs from malformed strings and incorrect string-handling. ESEC/FSE 2015, pages 555–566, New York, NY, USA, 2015.

[71] Jérémy Buisson and Fabien Dagnat. Recaml: Execution state as the cornerstone of reconfigurations. *SIGPLAN Not.*, 45(9):27–38, September 2010.

[72] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[73] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3):11:1–11:46, June 2012.

[74] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 101–110, New York, NY, USA, 2011. ACM.

[76] Gilad Mishne and Maarten de Rijke. Source code retrieval using conceptual similarity. RIAO '04, pages 539–554, Paris, France, France, 2004.

[77] Shaowei Wang, David Lo, and Lingxiao Jiang. Active code search: Incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 677–682, New York, NY, USA, 2014.

[78] Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M. German, and Katsuro Inoue. A method to detect license inconsistencies in large-scale open source projects. MSR '15, pages 324–333, Piscataway, NJ, USA, 2015.

[79] Thomas A. Alspaugh, Hazeline U. Asuncion, and Walt Scacchi. Analyzing software licenses in open architecture software systems. In *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '09, pages 54–57, Washington, DC, USA, 2009. IEEE Computer Society.

[80] John Long. Software reuse antipatterns. *SIGSOFT Softw. Eng. Notes*, 26(4):68–76, July 2001.

[81] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[82] Justin Pombrio and Shriram Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. *SIGPLAN Not.*, 49(6):361–371, June 2014.

[83] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. SWIM: synthesizing what i mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 357–367, 2016.

[84] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. Some from here, some from there: Cross-project code reuse in github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 291–301, Piscataway, NJ, USA, 2017. IEEE Press.

[85] D. Yang, P. Martins, V. Saini, and C. Lopes. Stack overflow in github: Any snippets there? In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 280–290, May 2017.

[86] Apache Commons. `https://commons.apache.org/`. Last accesed: 23-Jan-2018, 2018.

[87] Mukund Raghothaman, Yi Wei, and Youssef 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.

[88] Lee Martie, André van der Hoek, and Thomas Kwak. Understanding the impact of support for iteration on code search. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 774–785, New York, NY, USA, 2017. ACM.

[89] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, Jun 1997.

[90] Hui Fang, Tao Tao, and ChengXiang Zhai. A formal study of information retrieval heuristics. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, pages 49–56, New York, NY, USA, 2004. ACM.

[91] Wei Le and Shannon D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1047–1058, New York, NY, USA, 2014. ACM.

[92] Rylan Cottrell, Robert J. Walker, and Jörg 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*, SIGSOFT '08/FSE-16, pages 214–225, New York, NY, USA, 2008. ACM.

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

[94] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[95] Apache Commons Mathematics Library. `http://commons.apache.org/proper/commons-math/`. Last accesed: 23-Jan-2018, 2018.

[96] 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.

[97] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[98] 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.

[99] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, October 2017.

[100] Suresh Thummalapenta. Exploiting code search engines to improve programmer productivity. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 921–922, New York, NY, USA, 2007. ACM.

[101] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 362–373, New York, NY, USA, 2016. ACM.

[102] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina 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.

[103] Bei Shi, Wai Lam, Shoaib Jameel, Steven Schockaert, and Kwun Ping Lai. Jointly learning word embeddings and latent topics. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 375–384, New York, NY, USA, 2017. ACM.

[104] Rolf Jagerman, Carsten Eickhoff, and Maarten de Rijke. Computing web-scale topic models using an asynchronous parameter server. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, pages 1337–1340, New York, NY, USA, 2017. ACM.

[105] Suresh Thummalapenta and Tao Xie. Spotweb: Detecting framework hotspots via mining open source repositories on the web. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 109–112, New York, NY, USA, 2008. ACM.

[106] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[107] Reid Holmes. Do developers search for source code examples using multiple facts? In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE '09, pages 13–16, Washington, DC, USA, 2009. IEEE Computer Society.

[108] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 413–430, New York, NY, USA, 2006. ACM.

[109] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen 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.

[110] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[111] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 54–57, New York, NY, USA, 2006. ACM.

[112] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010.

[113] Andrea Lucia, Massimiliano Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Labeling source code with information retrieval methods: An empirical study. *Empirical Softw. Engg.*, 19(5):1383–1420, October 2014.

[114] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, March 2007.

[115] Lauren R. Biggers, Cecylia Bocovich, Riley Capshaw, Brian P. Eddy, Letha H. Etzkorn, and Nicholas A. Kraft. Configuring latent dirichlet allocation based feature location. *Empirical Softw. Engg.*, 19(3):465–500, June 2014.

[116] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, New York, NY, USA, 2006.

[117] George K. Baah, Andy Podgurski, and Mary Jean Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 189–200, New York, NY, USA, 2008.

[118] Yoshiki Higo and Shinji Kusumoto. How should we measure functional sameness from program source code? an exploratory study on java methods. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 294–305, New York, NY, USA, 2014.

[119] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Trans. Softw. Eng.*, 38(5):1069–1087, September 2012.

[120] David Shepherd, Kostadin Damevski, Bartosz Ropski, and Thomas Fritz. Sando: An extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 15:1–15:2, New York, NY, USA, 2012.

[121] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: A search engine for finding functions and their usages. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1043–1045, New York, NY, USA, 2011. ACM.

[122] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4):37:1–37:30, October 2013.

[123] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010.

[124] Doug Downey, Matthew Broadhead, and Oren Etzioni. Locating complex named entities in web text. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2733–2739, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[125] Openhub. `https://www.openhub.net/`, (accessed 2016-07-30).

[126] Lori L. Pollock, K. Vijay-Shanker, Emily Hill, Giriprasad Sridhara, and David Shepherd. Natural language-based software analyses and tools for software maintenance. In Andrea De Lucia and Filomena Ferrucci, editors, *ISSSE*, volume 7171 of *Lecture Notes in Computer Science*, pages 94–125. Springer, 2011.

[127] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 279–290, New York, NY, USA, 2014.

[128] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. pages 214–223, Nov 2004.

[129] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, pages 173–180, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[130] Xiaowen Ding, Bing Liu, and Lei Zhang. Entity discovery and assignment for opinion mining applications. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 1125–1134, New York, NY, USA, 2009.

[131] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014.

[132] Mark Gabel and Zhendong 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.

[133] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA, 2008.

[134] Peter C. Rigby and Martin P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 832–841, Piscataway, NJ, USA, 2013. IEEE Press.

[135] H. Schildt. *Java: The Complete Reference, Ninth Edition.* The Complete Reference. McGraw-Hill Education, 2014.

[136] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall software series. Prentice Hall, 1988.

[137] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. ACL '05, pages 363–370, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[138] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina 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.

[139] Di Yang, Aftab Hussain, and Cristina Videira 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.

[140] S. Subramanian and R. Holmes. Making sense of online code snippets. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 85–88, May 2013.

[141] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. Csnippex: Automated synthesis of compilable code snippets from q&a sites. In *ISSTA*, 2016.

[142] H. Sanchez and J. Whitehead. Source code curation on stackoverflow: The vesperin system. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 661–664, May 2015.

[143] Hao Zhong and Xiaoyin Wang. Boosting complete-code tool for partial program. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 671–681, Piscataway, NJ, USA, 2017. IEEE Press.

[144] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[145] Barthelemy Dagenais and Martin P. Robillard. Semdiff: Analysis and recommendation support for api evolution. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 599–602, Washington, DC, USA, 2009. IEEE Computer Society.

[146] Siddharth Subramanian and Reid 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.

[147] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 120–131, New York, NY, USA, 2016. ACM.

[148] Pavan Soni. *The Discovery of Grounded Theory (Glaser and Strauss, 1967).* 03 2015.

[149] StackOverflow. Stacktrace in stackoverflow. 2012.

[150] StackOverflow. Non java text in stackoverflow. `https://stackoverflow.com/questions/31457777`. Last accesed: 27-Jan-2018, 2015.

[151] StackOverflow. Line numbers in stackoverflow. `https://stackoverflow.com/questions/42202`. Last accesed: 27-Jan-2018, 2012.

[152] StackOverflow. Multiple blocks in stackoverflow. `https://stackoverflow.com/questions/23069036`. Last accesed: 27-Jan-2018, 2014.

[153] StackOverflow. Missing definitions in stackoverflow. `https://stackoverflow.com/questions/2097265`. Last accesed: 27-Jan-2018, 2010.

[154] Stackoverflow. `https://stackoverflow.com/questions/5152372`. Last accesed: 27-Jan-2018, 2017.

[155] StackOverflow. Ellipses in stackoverflow. `https://stackoverflow.com/questions/7024879`. Last accesed: 27-Jan-2018, 2011.

[156] StackOverflow. Ellipses and variable arguments in stackoverflow. `https://stackoverflow.com/questions/1656901`. Last accesed: 27-Jan-2018, 2015.

[157] StackOverflow. Psudocode in stackoverflow. `https://stackoverflow.com/questions/13954276/`. Last accesed: 27-Jan-2018, 2013.

[158] StackOverflow. Html elemens in stackoverflow. `https://stackoverflow.com/questions/22655193`. Last accesed: 27-Jan-2018, 2014.

[159] StackOverflow. Non java statements in stackoverflow. `https://stackoverflow,com/questions/8905028`. Last accesed: 7-Feb=2018, 2012.

[160] StackOverflow. Missing tokens in stackoverflow. `https://stackoverflow.com/questions/30409192`. Last accesed: 27-Jan-2018, 2015.

[161] StackOverflow. Java version issue in stack overflow. `https://stackoverflow.com/questions/26214693`. Last accesed: 27-Jan-2018, 2014.

[162] StackOverflow. Extra tokens in stackoverflow. `https://stackoverflow.com/questions/13631663`. Last accesed: 27-Jan-2018, 2012.

[163] Lori L. Pollock, K. Vijay-Shanker, Emily Hill, Giriprasad Sridhara, and David Shepherd. Natural language-based software analyses and tools for software maintenance. In Andrea De Lucia and Filomena Ferrucci, editors, *ISSSE*, volume 7171 of *Lecture Notes in Computer Science*, pages 94–125. Springer, 2011.

[164] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 309–320, New York, NY, USA, 2016. ACM.

[165] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 402–413, New York, NY, USA, 2014. ACM.

[166] Denzil Correa and Ashish Sureka. Chaff from the wheat: Characterization and modeling of deleted questions on stack overflow. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 631–642, New York, NY, USA, 2014. ACM.

[167] Fabio Calefato, Filippo Lanubile, Maria Concetta Marasciulo, and Nicole Novielli. Mining successful answers in stack overflow. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 430–433, Piscataway, NJ, USA, 2015. IEEE Press.

[168] Jiwoon Jeon, W. Bruce Croft, Joon Ho Lee, and Soyeon Park. A framework to predict the quality of answers with non-textual features. In *SIGIR*, 2006.

[169] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. Improving low quality stack overflow post detection. *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 541–544, 2014.

[170] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34, 2012.

[171] MohammadReza Tavakoli, Abbas Heydarnoori, and Mohammad Ghafari. Improving the quality of code snippets in stack overflow. In *SAC*, 2016.

[172] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Stormed: Stack overflow ready made data. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 474–477, 2015.

[173] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. Extracting structured data from natural language documents with island parsing. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 476–479, 2011.

[174] Yichen Xie and Dawson Engler. Using redundancies to find errors. *IEEE Trans. Softw. Eng.*, 29(10):915–928, October 2003.

[175] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 480–490, Washington, DC, USA, 2004. IEEE Computer Society.

[176] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 297–308, New York, NY, USA, 2016.

[177] Shuktika Jain. Automated generation of programming language quizzes. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 1051–1053, New York, NY, USA, 2015.

[178] Jin Zhao, Min-Yen Kan, and Yin Leng Theng. Math information retrieval: user requirements and prototype implementation. In *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, pages 187–196. ACM, 2008.

[179] Content Markup. `https://www.w3.org/TR/MathML3/chapter4.html`. Last accesed: 23-Jan-2018, 2018.

[180] Ridhi Jain, Sai Prathik, Venkatesh Vinayakarao, and Rahul Purandare. A search system for mathematical expressions on software binaries. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 487–491, New York, NY, USA, 2018. ACM.