



AFRL-RI-RS-TR-2018-203

MINING DEVELOPER COMMUNICATIONS TO CREATE A WEB-SCALE REPOSITORY OF DOCUMENTED AND ANALYZABLE SNIPPETS

ABB, INC.

AUGUST 2018

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2018-203 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

STEVEN DRAGER
Work Unit Manager

/ S /

JOHN MATYJAS
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) AUGUST 2018		2. REPORT TYPE FINAL TECHNICAL REPORT		3. DATES COVERED (From - To) SEPT 2016 – MAR 2018	
4. TITLE AND SUBTITLE Mining Developer Communications to Create a Web-Scale Repository of Documented and Analyzable Snippets				5a. CONTRACT NUMBER FA8750-16-2-0288	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 61101E	
6. AUTHOR(S) David Shepherd				5d. PROJECT NUMBER MUSE	
				5e. TASK NUMBER SA	
				5f. WORK UNIT NUMBER BB	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ABB Inc. 305 Gregson Dr. Cary, NC 27511				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI	
				11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2018-203	
12. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Typical software developer communications, such as chat conversations or forum posts, contain both code snippets and natural language text that describes those code snippets. Thus, developer communications on the web are an important resource for large-scale mining of information about the functionality, quality, and other properties of code snippets. Consistent with the goals of the DARPA Mining and Understanding of Software Enclaves (MUSE) program, the goal of this research project was to enable targeted access to the software development knowledge captured in the code snippets, as well as the natural language text describing those code snippets, embedded within developer communications. Key research accomplishments that stem from this project include an understanding of what information about code snippets is available in different kinds of developer communications, an in-depth analysis of two kinds of developer communication to compare their efficacy in supporting software engineering tools, a new technique to extract the available information from a specific kind of developer communication, and a technique to enable web-scale code clone detection and search, and ultimately, curation of documented, analyzable code snippets extracted from developer communications.					
15. SUBJECT TERMS Mining and understanding software enclaves, mining software repositories, developer communications, software engineering tools					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 65	19a. NAME OF RESPONSIBLE PERSON STEVEN DRAGER
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) N/A

TABLE OF CONTENTS

Section	Page
LIST OF FIGURES	iii
LIST OF TABLES.....	iv
1.0 SUMMARY.....	1
2.0 INTRODUCTION	3
2.1 Understanding What Information about Code Snippets Is Available in Different Kinds of Developer Communications	3
2.2 Comparing the Efficacy of Two Kinds of Developer Communications in Supporting Software Engineering Tools	3
2.3 Extracting Code Snippets and Their Descriptions from Research Articles.....	5
2.4 Enabling Web-scale Code Clone Detection and Search.....	6
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES.....	8
3.1 Understanding What Information about Code Snippets Is Available in Different Kinds of Developer Communications	8
3.2 Comparing the Efficacy of Two Kinds of Developer Communications in Supporting Software Engineering Tools	11
3.3 Extracting Code Snippets and Their Descriptions from Research Articles.....	16
3.4 Enabling Web-scale Code Clone Detection and Search.....	20
4.0 RESULTS AND DISCUSSION	25
4.1 Understanding What Information about Code Snippets Is Available in Different Kinds of Developer Communications	25
4.2 Comparing the Efficacy of Two Kinds of Developer Communications in Supporting Software Engineering Tools	29
4.3 Extracting Code Snippets and Their Descriptions from Research Articles.....	35
4.4 Enabling Web-scale Code Clone Detection and Search.....	42
5.0 CONCLUSIONS	48
5.1 Understanding What Information about Code Snippets Is Available in Different Kinds of Developer Communications	48
5.2 Comparing the Efficacy of Two Kinds of Developer Communications in Supporting Software Engineering Tools	48

5.3	Extracting Code Snippets and Their Descriptions from Research Articles	49
5.4	Enabling Web-scale Code Clone Detection and Search	49
6.0	REFERENCES	51
	APPENDIX.....	57
	LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS	58

LIST OF FIGURES

Figure	Page
Figure 1. Methodology Procedure	8
Figure 2. Annotation Example.....	9
Figure 3. Overview of CoDesNPub Miner	17
Figure 4. Delta Inverted Index Data Structure for CB ₁ to CB ₅	23
Figure 5. Heatmap of the Frequency of Information Types in Different Document Kinds	27
Figure 6. Percentage Sentences of Each Type in Slack _{manual}	33
Figure 7. Frequency of Information in Research Articles by Label	39
Figure 8. Frequency of Information in Research Articles by Sub-label	40
Figure 9. Relative Frequency of Each Feature Indicating Code Description Text.....	41
Figure 10. Comparison of Execution Time for Different Input Sizes	43
Figure 11. Comparison of Number of Candidate Pairs (10,000 Files)	45

LIST OF TABLES

Table	Page
Table 1. Subjects of Study	9
Table 2. Data Sets and Sizes	13
Table 3. Measures for First Research Question	14
Table 4. Measures for Second Research Question.....	16
Table 5. Code Snippet and Description Availability by Document Kind.....	26
Table 6. Description of Labels and Sub-labels with Example Cues in Text	28
Table 7. Findings for First Research Question, Part 1 (Min < Median < Max)	29
Table 8. Findings for the First Research Question, Part 2 (Min < Mean < Max).....	30
Table 9. Statistical Results for the First Research Question (Clojure)	30
Table 10. Indicators of an Accepted Answer in Slack _{manual}	31
Table 11. Findings for the Second Research Question	32
Table 12. Precision of Seed Heuristics	37
Table 13. Effectiveness of Code Description Identification	37
Table 14. Examples of Analyzed Code Description Sentences.....	38
Table 15. Comparison of Execution Time in Seconds (10,000 Files)	44
Table 16. Performance of Code Clone Search.....	47

1.0 SUMMARY

Developer communications, such as those found in blogs, emails, Question & Answer (Q&A) forums, or in bug trackers, contain a wealth of information from developer experience, including varied examples of how to achieve specific coding goals or of how to use various application programming interfaces (APIs) appropriately. Because developer communications typically contain a mixture of code snippets and natural language text describing those code snippets, they provide the opportunity to learn from what developers are saying or asking about the code snippets under discussion. Large corpora of developer communications provide the opportunity to automatically learn common API behaviors, properties, and vulnerabilities, as well as to complement similar information learned by mining software repositories. Previous research addressed questions about the information available in specific developer communications (such as code reviews or tutorials) but did not address questions about the code snippets embedded in those developer communications.

Consistent with the goals of the Defense Advanced Research Projects Agency (DARPA) Mining and Understanding Software Enclaves (MUSE) program, the overall goal of this research and development project was to enable targeted access to the software development knowledge captured in the code snippets, as well as the natural language text describing those code snippets, embedded within developer communications. Our key research accomplishments in this project were to: (1) catalog the different kinds of developer communications that may contain code snippets and understand what information about code snippets is available in the cataloged kinds of developer communications, (2) conduct an in-depth comparison of two kinds of developer communication to compare their efficacy in supporting software engineering tools, (3) develop a new technique to extract the available information from a specific kind of developer communication, and (4) develop a technique to enable Web-scale code clone detection and search (and ultimately curation of documented, analyzable code snippets extracted from developer communications).

This report is based on earlier published works [1], [2], [3] describing the key research accomplishments of this project.

The findings related to our first research accomplishment [1] include that there are wide variations in the numbers of code snippets, mean size of code snippets and mean lines of natural language text across the 12 different kinds of developer communications that we considered. We also found that information about origin and clarity of code snippets is rarely available in any of the studied developer communications. Information about efficiency is found almost exclusively in bug reports, while testing information was found almost exclusively in code reviews. We also used human annotators to code observations about the types of information available in the natural language text that describes embedded code snippets. Based on their samples, we note that sometimes individual words are adequate cues for a given kind of information (e.g., a name of a programming language or framework), while sometimes a cue requires a phrase, or sequence of words (e.g., “see how much cleaner it is” or “hard to parse”). For most types of information, phrases are needed to cue the kind of information available in the text. The structure and content

of the phrases differ across information types, which suggests that automating the detection of the kind of information embedded in the natural language text about a code snippet is not trivial. Detection techniques require the combination of natural language processing and machine learning.

The findings related to our second research accomplishment concern the information found in developer chat conversations and question and answer posts on Stack Overflow. Chat conversations generally provide less information of interest for mining-based software tools (e.g., code snippets, API mentions, and software-specific terms) than do Q&A posts. However, chat conversations provide more links to external information and more bad/negative code examples, both of which could be useful for mining-based software engineering tools. We also found that questions posted to Slack are about as likely as questions posted to Stack Overflow to receive either no response or to have an accepted answer. Yet, identifying an accepted answer is non-trivial and requires both identifying the responsive sentence(s) and the question to which those sentences are responsive. Part of this challenge stems from the free-form style of chat conversations, in which a question may be followed by a series of clarification or follow-up questions, and contextual clues must be followed to determine which question is ultimately being answered. Our other observations about developer chats on Slack include that these channels serve as clearinghouses for other information available on the Web, and as such they can be mined to provide more context to other data sources or even as comments on the quality of other data sources. We also observed that developers use Slack to share opinions on best practices, which Stack Overflow explicitly forbids. The availability of such information may lead to new mining opportunities for software engineering tools.

The third research accomplishment [2] is the first technique to automatically identify natural language descriptions of code snippets embedded within technical manuscripts, where code snippets can be separated as figures that are not located next to their descriptive text. Analysis of the descriptive text demonstrates the variety of information about the embedded code snippets that could be learned. The fourth research accomplishment [3] is a novel code clone detection technique based on adaptive prefix filtering. The design of the approach enables not only Web-scale code clone detection, but also code clone search, which is in contrast to previous Web-scale code clone techniques.

For each of the four research accomplishments, this report provides motivation, a detailed description of the technical approach, a summary of the research results, and a discussion of the implications of those results. The appendix of this report lists the publications and presentations for this project.

2.0 INTRODUCTION

Each heading in this section corresponds to the motivation for and overview of a specific research accomplishment. In Section 3.0 we provide a detailed description of the technical approach that we used in this project to achieve our research goals. In Section 4.0 we report our research results and discuss the implications of those results. Finally, in Section 5.0 we conclude with a summary of the project accomplishments and an overview of future work. A complete list of references is available in Section 6.0.

2.1 Understanding What Information about Code Snippets Is Available in Different Kinds of Developer Communications

Researchers have developed analyses to mine various information from the natural language text surrounding code snippets in emails [4], bug reports [5], Q&A forums [6], [7], and tutorials [8]. For example, Panichella et al. [5] developed a feature-based approach to automatically extract method descriptions from bug tracking systems and mailing lists. Vassallo et al. [6] built on their previous work [5] to design a tool called CODES that extracts candidate method documentation from Stack Overflow discussions and creates Javadoc descriptions. Wong et al. [9] mine comments from Stack Overflow by extracting text preceding code snippets and using the code-description mappings in the posts to automatically generate descriptive comments for similar code snippets matched in open-source projects.

Collectively, the aforementioned efforts demonstrate the potential for extracting and using information embedded in natural language text of developer communications for software engineering tools. They also raise the questions of what other kinds of information can be extracted from these particular kinds of developer communications and of what types of information can be extracted from other kinds of developer communications.

Our work in this area provides indications regarding the kinds of developer communications that would serve as the most fruitful sources for different types of information, and also provide insights into what cues might help in automatically extracting that information.

2.2 Comparing the Efficacy of Two Kinds of Developer Communications in Supporting Software Engineering Tools

Researchers have mined the knowledge embedded in Q&A forums such as Stack Overflow for supporting integrated development environment (IDE) recommendation [10], [11], [12], [13], [14], [15], learning and recommendation of APIs [16], [17], [18], automatic generation of comments for source code [9], [19], and in building thesauri and knowledge graphs of software-specific terms and commonly-used terms in software engineering [20], [21]. These successes

suggest that other kinds of developer communications may also serve as information sources for mining-based software engineering tools.

Developer chat conversations take place in the increasingly utilized public chat communities, open to developers interested in discussing a specific programming language or technology, including communities hosted on Slack, Hipchat, Gitter, internet relay chat (IRC), Microsoft Teams, and Freenode. Developers use these communities to ask and answer specific development questions, with the aim of improving their own skills and helping others. Over 1 million active users participate per week on Slack, the most popular platform for these public chat communities, hosting numerous active public channels focused on software development technologies [22].

While chat communities share some commonalities with other developer communication channels, they also differ in intent and use. Overall, Q&A forum content is archival, while chat community content is transient. Q&A forums encourage longer, more in-depth questions that receive one or more well-thought-out answers and are subsequently read by many developers. Developer chats are typically informal conversations, with rapid messages between two or more developers, where several clarifying questions and answers are typically communicated in short bursts. Chats thus contain shorter, quicker responses, often interleaved with non-information-providing messages. The chat conversations are usually short-lived, often available only for a relatively short period of time (e.g., a week or less for Slack channels with a 10,000 message limit), and similar questions can be repeated in the channel many times.

Most studies of developer chat communities have focused on learning about how they are used by development teams and for analyzing developer behaviors and interactions [23], [24], [25], [26], [27], [28]. Limited work has focused on mining and extracting specific types of information from developer chats. Alkadhi et al. [23] examined the frequency and completeness of available rationale in HipChat messages, and the potential of automatic techniques for rationale extraction. To the best of our knowledge, none of the previous work has investigated chat communities from the perspective of understanding how fruitful the chat communities would be for mining valuable nuggets of information — similar to those that have been mined from other sources — for supporting software tool development.

As part of this project, we investigated the potential usefulness and challenges of mining chat communities for supporting software engineering tasks and tools. We conducted an exploratory study to compare the content in Q&A-focused public chat communities (e.g., Slack) with Q&A-based discussion forums (e.g., Stack Overflow), because both resources are believed to share similar intent of learning and sharing information among developers. We explored the availability and prevalence of information in chat conversations, which provides us with the first insight into the promise of chat communities as a mining source. Additionally, we studied the characteristics of chat communities to investigate the feasibility of applying automatic information extraction techniques (previously developed for Q&A forums) on chat communities. We analyzed information such as characteristics of embedded code snippets and of the text describing the code snippets.

The overall goal of this study was to gain insight into whether it is worth mining chat communities for information to support software engineering tools, and whether there are additional opportunities from mining chat communities that are not offered by Q&A forums. Our results indicate the prevalence of useful information, and thus provide insights that could aid future research in exploiting the various kinds of unexplored information available in developer chats.

2.3 Extracting Code Snippets and Their Descriptions from Research Articles

Mining code snippets and their descriptions from research articles presents challenges beyond those faced in mining from unstructured documents such as forums, bug reports, emails, and issue tracking. In all of these unstructured documents, including research articles, the code snippets are intermixed with natural language text, sometimes separated by blank lines and sometimes single code statements within paragraphs or even individual identifiers within sentences. In all of these documents, the code snippets are embedded in the mainstream text. In contrast, code snippets in research articles are sometimes embedded within the text, but often separated as figures, which are rarely positioned in the flow consecutively with the text that describes them. The figure could be located in a different section or different page. This physical separation of code snippet from description makes the description identification problem, i.e., the problem of identifying all of the embedded code snippets, more difficult. The problem is further complicated by the common situation in which the research article contains multiple code snippets, in which case the textual descriptions that are identified need to be mapped to the corresponding code snippet. Lastly, the description identification problem would be simpler when there is less text to analyze, but unfortunately research conference articles in general contain 400 lines of natural language text, considerably longer than emails, bug reports, and forum entries. If one wants to scale a code snippet and description mining technique, the technique cannot analyze every line of text.

Researchers have developed techniques to automatically extract code snippets from emails, bug reports, Q&A sites and tutorials [4], [29], [30], [31]. These techniques are also applicable to extract code snippets from research articles. Less work has focused on the code description identification problem. Traceability analyses can use code terms in a sentence or paragraph as an indicator of which application programming interface is being described [30], [32]. Text preceding code snippets in Stack Overflow can be extracted as potential comments for similar code snippets in an application [33]. Similarly, method descriptions can be extracted using clues in the text [8], [33].

To the best of our knowledge, our work is the first to address the code description identification problem for research articles and other similar documents such as dissertations. The main contributions of our work in this area are:

- Heuristics to automatically identify and map text that is describing code snippets in research articles, including snippets embedded as figures
- Heuristics to expand the neighborhood of identified descriptions to include informative, yet less obviously related text

- A tool that takes research articles as input and outputs an extensible markup language (XML)-based representation with markups to associate identified code snippets with their corresponding descriptions
- A study that evaluated the effectiveness of the presented code description identification techniques

2.4 Enabling Web-scale Code Clone Detection and Search

Performing code clone detection across numerous software repositories is a common use case. Specific applications for large scale code clone detection include querying library candidates [34], categorizing copyright infringement and license violations [35], [36], plagiarism detection [35], [36], finding product lines in reverse engineering [36], [37], tracing the origin of a component [38], searching for code blocks in large software repositories [39], [40], and spotting analogous applications in Android markets [41], [42]. However, most existing code clone detection techniques have difficulty scaling up to extremely large collections of source code [43], [44].

Among the tools aimed towards large scale code clone detection, a common limitation is in the complexity of differences among clones they can detect. For instance, scalable token based approaches [34], [45], [46] have difficulty detecting near miss (Type-3) code clones, which can occur more frequently than other types of clones [44], [47], [48]. Parallel and distributed clone detection techniques like D-CCFinder [49] can be more burdensome to manage, requiring specialized hardware or software support, while tree based code clone detection techniques, such as Deckard [50], place higher demands on memory.

As part of this project, we proposed a token-based code clone detection technique aimed at scalability and detecting Type-3 clones, consisting of two main steps: filtering and verification. In the filtering step we aim to significantly reduce the number of code blocks for comparison, removing from consideration blocks that do not have any possibility of being code clones. In the verification step we determine whether candidate pairs that survived the filtering step are really code clones. This two-step process greatly reduces the runtime of code clone detection, allowing the technique to scale up to very large corpora. This technique is based on the adaptive prefix filtering heuristic [51], which is an extended version of prefix filtering heuristic [52], [53] previously applied towards code clone detection in SourcererCC [42]. To our knowledge, SourcererCC was the best scaling code clone detection tool able to detect Type-3 clones. We demonstrated improvements in execution time relative to SourcererCC, while obtaining the same accuracy, for many common similarity thresholds, when evaluated on a large scale inter-project source code corpus [54].

A separate novel idea that we conceived as part of this project is the effective application of our technique to code clone search, without modification, in addition to code clone detection. Code clone search is a related problem to code clone detection where the user specifies a single code block (i.e. query block) to search for in a corpus of many code blocks. Once indexed, the corpus

should be able to serve numerous such queries. Ours is among few techniques that can be applied to both of these problems. Thus, the contributions of our work in this area are:

- A novel code clone detection technique that can scale to very large scale source code repositories (or sets of repositories) with the ability to detect Type-1, Type-2, and Type-3 code clones, while maintaining high precision and recall
- An extension of our proposed technique so that it can be effectively utilized for code clone search without modification

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

In this section we provide a detailed description of the technical approach that we used in this project to achieve our research goals. The section is organized by research accomplishment. That is, each heading in this section corresponds to the methods, assumptions, and procedures used for a specific research accomplishment.

3.1 Understanding What Information about Code Snippets Is Available in Different Kinds of Developer Communications

We conducted a study [1] to investigate the following research questions:

- What are the characteristics of the code snippets embedded in different kinds of developer communications?
- What type of information is available in each developer communication kind? Which kinds provide similar information? Which information is most readily available across all kinds of developer communications?
- What are the cues that indicate code snippet related information? How do the cues differ across different developer communication kinds? What influential words/phrases indicate the different types of information in different developer communications?

Figure 1 depicts the major steps of our study. After selecting a set of documents of each developer communication kind under investigation, we redacted the code snippets, and asked human annotators to make as many observations about what they could learn about the missing code snippets strictly from the text and also to highlight the text on which they based their observations. We then developed a labeling scheme to code the observations and analyzed both the codings and other information gained from the frequency and sizes of the code snippets across different kinds of developer communications. Figure 2 provides an illustrative example of steps 1–3.

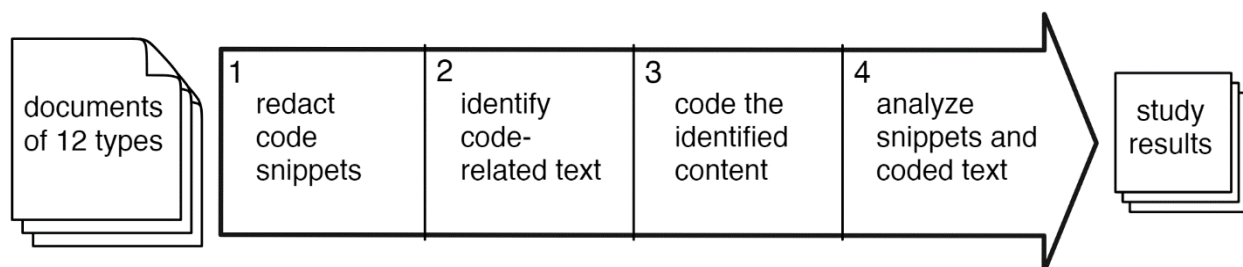


Figure 1. Methodology Procedure

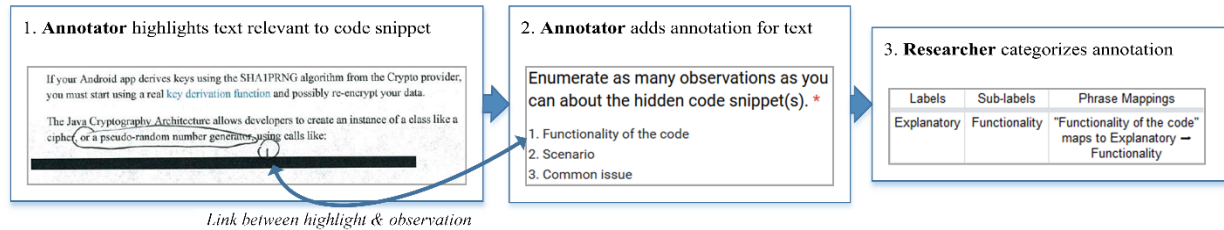


Figure 2. Annotation Example

Table 1 specifies the 12 kinds of developer communications that we studied, including example sources from which we sampled communications and the unit of granularity that we studied. We collected 60 instances, with at least two instances for each kind. We randomly selected instances from 51 distinct sources, all well known, popular sites or high-profile projects. We excluded instances that had no code snippets.

Table 1. Subjects of Study

Developer Communication	Example Origin	Unit of Analysis
Blog posts	MSDN	Blog post
Benchmarks	OpenMP	Chapter/section
Bug reports	Bugzilla	Bug report
Code reviews	Gerrit	Pull request
Course materials	cs.*.edu	PDF
Documentation	readthedocs.org	Page
E-books	WikiBooks	Chapter
Mailing lists	lkml.org	Conversation thread
Papers	IEEE Xplore	Paper
Presentations	SlideShare	Slide deck
Public chats	Slack	Conversation
Q&A sites	Stack Overflow	Post

After assembling the corpus, we redacted all multi-line code snippets from each instance. We redacted the code snippets because we wanted to determine what could be learned about them from only the contextual information in the instances. We redacted the code snippets using the redaction tools in Adobe Acrobat. Annotators were given documents with each line of a code snippet blacked out or covered entirely with a black rectangle. We did not redact small code snippets that were embedded within paragraphs of text.

We assigned 37 instances to 35 annotators. The annotators comprised 20 undergraduate, 13 graduate, and 2 professional researchers, all with prior programming experience. We assigned each annotator 3 instances to review. All of the instances were assigned to multiple annotators to review. We received 59 responses from 19 of the annotators (8 undergraduate, 11 graduate, 2 professional). The annotators submitted 49 reviews of 31 distinct instances, constituting 2-4 instances of each kind. 14 of these instances were annotated by multiple people, and in these cases we merged the observations from different annotators.

Each annotator was asked to enumerate as many observations as possible about the redacted code snippets. They also indicated which code snippet they believed was being described when there were multiple redacted code snippets in the same instance. They highlighted the text from which they made each observation. Thus, they created an implicit mapping between code snippet, related text, and observation from that text. The annotators made 322 observations in total.

Two of the authors identified core code properties that were mentioned throughout the annotators' observations. We defined eight major categories of labels, or codes, for the observed code properties. We further defined subcategories for each of these labels, which we call sublabels, to provide more detailed categorization of the observations for qualitative analysis. Table 6 presents the eight labels and their sublabels that we established. Finally, two authors coded each annotator observation with the labels and sublabels based on the occurrence of specific cues embedded in key phrases or key words.

Figure 2 gives an example of an instance being analyzed from start to finish. In step 1, an annotator highlights the parts of the instance that they think are related to the redacted code snippet. The annotator numbers each highlighted text segment. In step 2, the annotator writes and numbers observations for each highlight. In step 3, we determine a label, sub-label, and the phrases or words in the original highlighted text associated with the observation that cued the labeling.

To compare the number and size of code snippets in each instance kind, the authors manually counted the number of code snippets in each instance and the non-empty lines of code (LOC) per code snippet. Since instance kinds may vary in the length of a text line, we computed a normalized measure. We first took a sample from each instance kind and counted the number of characters (including white space and punctuation) to identify the range of number of characters per text line. We then automatically counted the number of characters in the whole instance, and divided by an average character count per line for that instance kind.

We performed two analyses of the coded text and annotators' observations. To determine the kinds of information available in each instance kind, we computed the frequency of occurrence of the different labels and sublabels. To gain an insight into how the different information is provided for each label, we examined the text highlighted by the annotators to identify the cue words or phrases that most likely triggered the annotator to highlight that information.

3.2 Comparing the Efficacy of Two Kinds of Developer Communications in Supporting Software Engineering Tools

We designed and conducted a study to answer the following research questions:

- Is the same information successfully mined in Q&A forums for supporting mining-based software engineering tools available and prevalent in chat communities?
- Which characteristics of chat communities may inhibit automatic mining of valuable information?

To answer our research questions, we curated datasets from both chats and Q&A forums. The first decision was what constitutes a “document” for analysis in each of these communities.

In a Q&A forum, we used a post as the basis for a document. A Q&A post can include questions, multiple answers, votes, comments, and other information. For our analysis, we defined a document to be the question and the two most popular answers (based on votes). We omitted further, less relevant, answers to the question which are likely to be either too similar to the first two answers, and therefore unnecessary to consider, or too different, and therefore likely to be noise. While the two most popular answers often includes the accepted answer, this isn’t always the case. In Stack Overflow, the original questioner indicates which answer is the accepted one. If the vote total differs from this, it is because other users have found a different answer more useful or because the context around the question has changed (necessitating a new answer).

The most reasonable equivalent to a Q&A forum post within a chat community is a single conversation that follows a Q&A format with someone seeking information and others providing answers in chat form. However, messages in chats form a stream, with conversations often interleaving such that a single conversation thread is entangled with other conversations, thus requiring preprocessing to separate, or disentangle, the conversations for analysis.

The disentanglement problem has been studied before in the context of IRC and similar chat platforms [55]. We leveraged the technique proposed by Elsner and Charniak that learns a supervised model based on a set of features between two chat messages [56]. The features include the elapsed time between the message pair, whether the speaker in the two messages is the same, occurrence of similar words, use of cue words (e.g., hello, hi, yes, no), and the use of technical jargon. For the training set, we manually annotated a set of 500 messages from each channel and constructed a model specific to each channel.

After we observed that some Slack channels can become dormant for a few hours at a time, we modified Elsner and Charniak’s algorithm to compute its features on the last five messages regardless of elapsed time. Without this modification, the technique splits messages that occurred with a gap of a few minutes into separate conversations. We also followed the procedure prescribed by Elsner and Charniak to create a set of technical words for the model by extracting all of the words occurring in Stack Overflow documents tagged with a particular tag that do not co-occur in the English Wikibooks corpus.

We established several requirements for dataset creation to reduce bias and threats to validity. In order to conduct an effective comparison between Q&A forums and chat communities, we needed to identify groups that primarily discussed software development topics, had a substantial collection of participants, and had a presence on both kinds of systems.

For this study, we chose Stack Overflow and Slack as modern, popular instantiations of Q&A forums and chat communities, respectively. We selected four programming communities with an active presence on both systems. We also focused on chats that followed a Q&A form.

Because programmatic access to the data in Slack communities is controlled by the administrators of the Slack team, we contacted several public Slack teams and asked for an API token that would allow us to read and store their data. Public Slack teams typically use Slack's free tier, which only stores the most recent 10,000 messages. Thus, for each Slack community that we selected for our study, we downloaded all of the discussion data from each channel every day for 74 days. Since Stack Overflow regularly publishes all of its data in extensible markup language format via the Stack Exchange Data Exchange, we extracted posts for this study from the 27-August-2017 release of the Stack Exchange Data Dump [57].

To enable comparison study between related communities on Stack Overflow and Slack, we first selected Slack programming communities and then identified related Stack Overflow communities by extracting all of the posts tagged with the language name from the Stack Exchange Data Dump. Table 2 shows the programming language communities used as subjects of study. A key problem in comparing Slack and Stack Overflow stems from the difference between the small timespan of Slack data we gathered, relative to Stack Overflow, which is archival and contains posts that span several years. Our concern was that the Slack conversations may not contain the same distribution of subtopics within a particular technology (e.g., Django or Flask within Python) as Stack Overflow. To address these concerns, we curated our comparison dataset by organizing the extracted Slack conversations and Stack Overflow posts by similar subtopics, using the following workflow: (1) remove uniform resource locators (URLs) and code snippets from the Slack and Stack Overflow datasets, (2) train a latent Dirichlet allocation (LDA) topic model using the larger Stack Overflow data, (3) remove background topics, which are very strongly expressed in the corpus (strongest topic for > 50% of documents) and do not express technical terms, (4) infer topic distributions for each document in Stack Overflow and conversation in Slack, (5) select the most strongly expressed topic for each document, and (6) compute comparative measures on documents that express the same topic in Stack Overflow and Slack.

We refer to the comparison datasets as SO_{topic} and $Slack_{topic}$. Table 2 includes the number of conversations from each channel and number of posts from each Stack Overflow community. For example, Table 2 shows that $Slack_{topic}$ consists of 8,843 conversations, while SO_{topic} contains 825,294 posts.

Some of the measures we wanted to use to investigate the potential use and challenges of mining chat communities are not easily automated with high accuracy. Thus, we created subsets of $Slack_{topic}$ and SO_{topic} which we call $Slack_{manual}$ and SO_{manual} , respectively. Since several of our measures involve the analysis of code snippets, we specifically chose documents that contain

embedded code snippets, particularly multiline code snippets, since they are the prime focus of most code snippet-related mining in other software artifacts. Although error and exception logs are often considered to be code snippets, for this study, we excluded them. For each dataset, we set a target of 100 multiline code snippets across all documents in each community. For *Slack_{manual}*, we selected 53 documents. For *SO_{manual}*, we selected 24 documents. *SO_{manual}* contains fewer documents than *Slack_{manual}* because each post tended to be longer and thus contained more code snippets.

Table 2. Data Sets and Sizes

Community	# Conversations		Community	# Posts	
	<i>Slack_{topic}</i>	<i>Slack_{manual}</i>		<i>SO_{topic}</i>	<i>SO_{manual}</i>
clojurians#clojure	1,497	15	clojure	13,920	7
elmlang#beginners	2,671	7	elm	1,019	5
elmlang#general	2,654	5	-	-	-
pythondev#help	1,657	13	python	806,763	5
racket#general	364	13	racket	3,592	7
Total	8,843	53	Total	825,294	24

To answer the question “is the same information successfully mined in Q&A forums for supporting mining-based software engineering (SE) tools available and prevalent in chat communities?” we focused on information that has been commonly mined in other software artifacts. Specifically, we analyzed code snippets, links to code snippets, API mentions, bad code examples, and software specific term frequency. To provide context, we also collected data on document length in Slack and Stack Overflow.

The text of each Slack document is stored in the Markdown text format. For the measures involving code snippets or links, we used regular expressions to extract the code snippets and the URLs from each conversation. The text of each Stack Overflow post is stored as hypertext markup language (HTML), thus, we used an HTML parser, BeautifulSoup [58], to extract the code and link elements from each post. We wrote Python scripts to analyze conversations and posts for all the automatically collected information shown in Table 3.

Document length is defined as the number of sentences in a document (i.e., a Slack conversation or a Stack Overflow post). We computed this measure on the non-code natural language text in each document using the sentence tokenizer from the Natural Language Toolkit (NLTK) [59].

Code snippet frequency is the number of code snippets per document, for which we counted all inline and multiline code snippets in a document. Code snippet length is the number of non-

whitespace characters in each code snippet. For multiline snippets, we also counted the number of non-blank lines in each snippet.

For link analysis, we counted all uniform resource locators in a document that match a particular service. For instance, for Gist link frequency, we counted all links to gist.github.com. For Stack Overflow link frequency, we counted all links to either stackoverflow.com or stackexchange.com. For the SO_{topic} dataset, we also included relative links in this measure.

We manually annotated the documents in each of the $Slack_{manual}$ and SO_{manual} datasets to calculate the measures related to API mentions and bad code examples. API mentions in code snippets and text are the numbers of unique APIs mentioned in the code snippets and in the non-code natural language text, respectively, for each document. Code snippets are considered to be bad examples if they are inefficient, do not implement the desired functionality or have poor readability or syntax errors. Bad code examples are often accompanied with negative words or phrases such as “ugly,” “error,” “does not work,” or “horrible.” We manually searched for the presence of such negative indicators in the natural language text to identify and count bad code examples in the documents.

Software-specific term frequency is defined to be the ratio of the number of software-specific terms to the total number of terms in a document. We followed the approach suggested by Elsner and Charniak [60] to create a list of software-specific terms for each community. In particular, for each community, we first created a list of all terms in our SO_{topic} dataset. From that list, we removed English stop words [61] and any term with length less than three. We then removed any term that appears more than once in the dump of the Wikibooks corpus. Finally, we removed any term that appears in only one, or in more than 80%, of the Stack Overflow posts for the given community. In the end, we obtained a list of software-specific terms for each community, and we used these lists to compute the measure for the respective communities.

Table 3. Measures for First Research Question

Measure	Definition	Datasets Used
Document length	# sentences in document	topic
Code snippet frequency	# code snippets in document	topic
Code snippet length	# characters in code snippet	topic
Gist link frequency	# links to Gist in document	topic
Stack Overflow link frequency	# links to Stack Overflow in document	topic
API mentions in code snippets	# unique APIs mentioned in code snippets	manual
API mentions in text	# unique APIs mentioned in non-code text	manual
Bad code examples	# erroneous code snippets	manual
Software-specific term frequency	% software-specific terms in document	topic

To answer the question “what characteristics of chat communities may inhibit automatic mining of valuable information?” we focused on measures that could provide some insights into the form of Q&A conversations (participant frequency, questions with no response, answer frequency) and measures that could indicate challenges in automation (how participants indicate accepted answers, questions with no accepted answer, natural language text context pertaining to code snippets, incomplete sentences, and noise within a document that suggest a need to filter. All measures were computed manually on the *Slack_{manual}* dataset, except participant frequency and questions with no response, which could be computed with high accuracy automatically with scripts. Manual analysis was performed by several of the authors. The measures that we collected to answer this research question are listed in Table 4.

To compute Participant frequency, we counted the number of unique users in each Slack conversation. For example, a conversation with 10 messages posted by 3 users has a Participant frequency of 3. We report the median, minimum, and maximum Participant frequencies over all Slack conversations in *Slack_{topic}*.

Questions with no response in *Slack_{topic}* was computed by counting the number of disentangled conversations that only consist of a single question and no other messages. As an indicator of a question, we rely on the message containing a question mark at its end.

For computing Answer frequency, we considered the number of unique different solutions proposed in response to the initial question in the conversation that explained the problem. Specifically, we computed the median, minimum and maximum number of unique answers over all the Slack conversations in *Slack_{manual}*. Uniqueness of the proposed solutions were determined by two of the authors.

Unlike Stack Overflow, Slack does not provide an option to accept a suggested solution in the platform. However, the correct solutions are often followed with positive emojis. In addition, there are prevalent textual clues that indicate that the suggested solution worked, and/or helped in solving the problem being discussed in the conversation. By manually analyzing all the natural language text in our dataset’s conversations, we built a list of words/phrases/emojis that participants used to indicate acceptance of answers - Indicators of accepted answers. We then counted the number of Questions with no accepted answer as the conversations where no clues for acceptance of answers were shown by the participants. The potential reasons for no acceptance of answers could be incorrectness/inefficiency of the suggested solutions, the user’s discretion to acknowledge and show acceptance, etc. We computed the percentage of conversations with no accepted answer by calculating the ratio of the number of conversations with no accepted answers, to the total number of conversations in our dataset.

Embedded code snippets in Slack conversations are surrounded by natural language text that describes various code metadata such as the functionality being implemented, data structures, code complexity and effectiveness, errors and exceptions. We analyzed all the sentences in the disentangled conversations to count the number of natural language sentences related to a code snippet – natural language text context. We specifically computed the median, minimum, and maximum number of sentences describing a code snippet.

Developer chat communications are informal by nature. Therefore, conversations include incomplete sentences, which potentially makes mining of essential information difficult. We used the Python NLTK [59] package to determine the incompleteness of the natural language sentences in the conversations. Specifically, we counted the total number of Incomplete sentences in a conversation, and report the overall percentage over all the analyzed Slack communities.

Similarly, conversations can include Noise, that is, sentences that do not provide useful information for mining such as ‘What do you mean?’, ‘Yes.’, ‘Any idea?’, etc. The annotators subjectively determined the noise/non-information sentences in a conversation. Then the total number of sentences in each conversation was counted to report the overall percentage of non-information sentences over all the analyzed Slack communities.

Table 4. Measures for Second Research Question

Measure	Definition	Datasets Used
Participant frequency	# participants in a conversation	<i>Slack_{topic}</i>
Questions with no response	% conversations that are a question only	<i>Slack_{topic}</i>
Answer frequency	# answers for a question	<i>Slack_{manual}</i>
Indicators of accepted answers	List of textual clues used as votes	<i>Slack_{manual}</i>
Questions without accepted answer	% conversations with no accepted answer	<i>Slack_{manual}</i>
Natural language text context	# natural language sentences about code	<i>Slack_{manual}</i>
Incomplete sentences	% incomplete sentences	<i>Slack_{manual}</i>
Noise in document	% non-information sentences	<i>Slack_{manual}</i>

3.3 Extracting Code Snippets and Their Descriptions from Research Articles

Figure 3 presents the phases of our whole process for automatic preprocessing, classification, identification and markup of research articles [2]. The input document is the research article, which may be a conference or a journal publication, in pdf format. The preprocessing phase renders the entire input document into plain text. Existing pdf-to-text converters can be used for this phase, such as pdftotext [62], convertmypdf [63], or convertpdftotext [64] making sure they can handle both single and double column formatting of articles. Since some code snippets are embedded into articles as images, this phase leverages existing image-to-text converters such as ocrconvert [65] or abbyyfinereader [66] to process the images into text so all code snippets can be identified.

The first text analysis phase partitions the content such that each block is classified into a single category of either source code or natural language text. Various tools are already developed by researchers to do similar content classification tasks [4], [5], [30], [31]. In our work for this project, we used a tool that takes the previously created plaintext as input, and outputs a tagged XML file with separate tags for natural language text and code snippets.

The main focus of our work for this project was the code description miner, which is comprised of two phases. The first phase identifies code-related seeds, which are natural language sentences that are directly related to an embedded code snippet through either the content or location of the natural language text. The second phase identifies natural language text neighboring code-related seeds that is highly likely to also be containing useful information about the code, but not directly identifiable without the seed text. We refer to these two kinds of code-related text as seeds and neighbors. We use linguistic and structural features of the text and embedded code snippets to identify and map the code descriptions to the associated code snippets in the document. The system can be implemented to output either as an XML version of the original article with the code snippets and related seeds and neighbors marked up, or a set of extracted code snippets and related seeds and neighbors for a database of mined code examples with descriptive information.

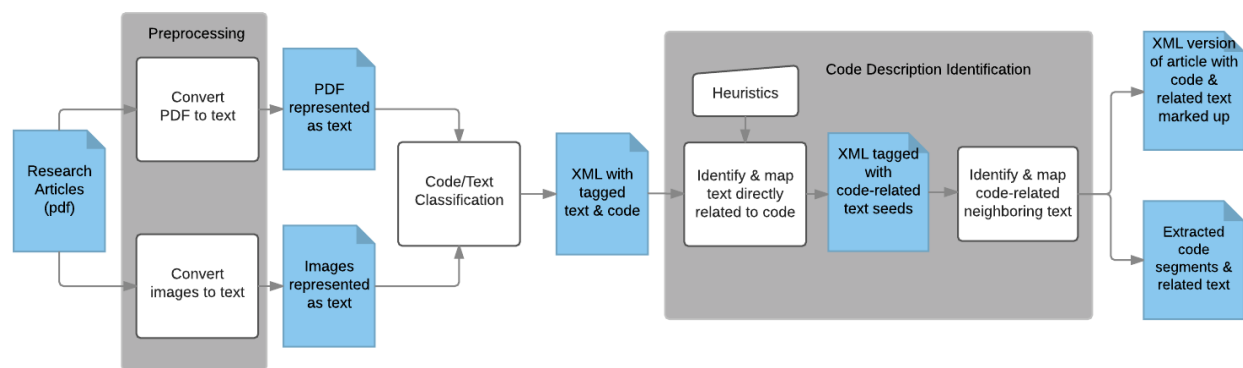


Figure 3. Overview of CoDesNPub Miner

To develop our automatic code description identification technique, we analyzed the text of randomly selected computer science research articles from the Association for Computing Machinery (ACM) and Institute of Electrical and Electronics Engineers (IEEE) digital libraries (DL), which collectively included over 200 code examples. Based on our manual inspection of both text related to the code snippets and text not related to the code snippets, we developed a set of heuristics that focus on features of sentences, including location and lexical and phrasal information. We first describe our individual heuristics for identifying sentences as code description and then describe how we combine the heuristics to perform code description identification.

While some code snippets in research articles are embedded within the running text, many are included as separate figures or listings. Typically, when code appears as a figure or listing, authors will refer the reader to the code they are discussing by using phrases such as “In Figure 1” or “Listing 1.” These references are accurate cues for an automatic system to identify sentences

related to the code snippet, when we are able to identify that the figure being referenced is indeed a code snippet. The *ReferencesCodeFigure* heuristic identifies sentences that contain the word “figure” or “listing”, and uses the figure or listing number reference to check whether the referenced figure has been classified as code.

Sometimes, authors of research articles place their code examples directly inline within the running text, similar to the use of code snippets in online forums and emails. When this occurs, it is most likely that they are discussing the code snippet in sentences just before or just after (or both before and after) the code snippet itself. The *TextBefore* and *TextAfter* heuristics identify the sentences immediately before and immediately after any inline code snippet as potential code descriptions, respectively. Without combining with other heuristics, these heuristics can be inaccurate if the author always describes code snippets before or after and not both locations. These heuristics capture the relative location only of sentences surrounding inline code. Additional sentences in the nearby location will be considered by the neighborhood sentence identification heuristics.

Sentences describing code snippets in a research article often contain code identifiers from the associated code snippet. The use of code identifiers is particularly common when describing the steps comprising the code, explaining the functionality of each statement or block. Thus, the *ContainsCodeIdentifiers* heuristic identifies all the code snippets in a research article that contain a word that also appears in any of the code snippets as a user-defined identifier. This heuristic requires tokenization of the code snippets within the document, creation of a dictionary of variable, method, and class names for each code snippet, and removal of keywords from those dictionaries. The heuristic identifies and maps sentences to associated code snippets based on occurrences of the dictionary names in the sentence. This heuristic has the potential to be inaccurate when code snippets use identifiers that are commonly used as regular words in sentences.

Authors sometimes use specific cue words or phrases pertaining to software engineering and development when describing code snippets in research articles. The *ReferencesCodeByPosition* heuristic identifies the sentences that have specific cue words or phrases that suggest that a sentence is describing a code snippet in the document. This heuristic looks for phrases such as “...in the following code...” or “in the running example”. Specifically, this heuristic aims to identify sentences containing code-indicating words such as ‘code’, ‘method’, ‘loop’, or ‘Javascript’. Based on the adjective in the phrase, it searches either before or after the sentence for a code snippet located in the designated relative position near the neighboring two paragraphs to the sentence, to confirm that the sentence is referring to a code snippet.

A given sentence may be identified as a potential code description sentence by more than one heuristic. For instance, a given sentence might say “In the code below,” and also be located immediately before a code snippet, or a sentence might mention a code identifier and include the phrase “In our example code.” A given sentence might contain more than one identifier which is a stronger indicator than just one identifier that might occur in more than one snippet.

We combine the heuristics by assigning a score to a sentence each time a heuristic indicates that it is potentially a code description. We pose two scoring schemes as follows:

- Equal scores: All cues are treated as equally contributing to the potential for a sentence to be a code description. Each instance of any heuristic being triggered for a given sentence results in adding a score of 1 to the total score for that sentence.
- Accuracy-based scores: Some heuristics such as *ReferencesCodeFigure* are highly likely to accurately identify a sentence as a code description whereas others could be less accurate, such as *TextBefore* and *TextAfter*. Thus, this scoring scheme assigns different scores to each instance of different heuristics depending on the basis of our observations of relative accuracy during our work with the development set. Based on our development set analysis, the final scores for each heuristic for the best scoring scheme are a score of 3 for *ReferencesCodeFigure*, a score of 2 each for *ContainsCodeIdentifiers* and *ReferencesCodeByPosition*, and a score of 1 for each of *TextBefore* and *TextAfter*.

Each heuristic is applied to the document, resulting in a score for each sentence. The heuristics can be applied in any order as their application order does not affect the final scoring. A threshold is used with the final scores to classify a given sentence as code description. In our experimental study, we performed a threshold analysis, including a threshold that requires only one heuristic to be triggered to consider a sentence as a code description, up through requiring some combination of heuristics that achieves a high score. We evaluated the various precisions and numbers of sentences identified with different thresholds and scoring schemes.

Our observations during development revealed that there may be neighboring text to the code description sentences identified by the heuristics that is also related, but the heuristics do not indicate that directly. The additional sentences often describe finer details, such as the intuition for implementing the code, and are important for better understanding and reuse of the same code example. For example, code-related neighboring text might be a sentence such as “This allows the attacker to circumvent the name check and get access to all user records in the database.” Identifying this sentence is important since it describes the consequence of the security vulnerability threat in the referenced example code, but the heuristics with the textual cues would not identify this sentence.

Our manual analysis suggested that if a part of a paragraph of text contains sentences identified by the heuristics to describe a code snippet, then the entire paragraph often describes the code snippet extensively. However, not every paragraph with an identified seed sentence was entirely a code description. Therefore, we explored several percentages of paragraph sentences as minimum numbers of seed sentences needed to consider the whole paragraph as code description text.

- At least one sentence in the paragraph matches one or more heuristics to identify text directly related to code.
- At least (25%, 50%, or 75%, respectively) of the total number of sentences in the paragraph matches one or more heuristics to identify text directly related to code.

Consider the following paragraph, which is typical of those found in software engineering research papers (e.g., [30]). We consider each heuristic on each sentence of this paragraph as our code description identification process works at sentence granularity. We describe the example using the accuracy-based scoring scheme.

Fig. 1 illustrates a method that exercises two program features. The method uses the basic functionality of the API class CredentialCache, including the methods add, get, remove, and clear. The method exercises the two features on lines 2-3 and lines 6-10, respectively. The features share an object, cache. The features use different method invocation sequences, but they do share a common subsequence of method invocations. In particular, the invocations of get and getFullyQualifiedHostName are used to exercise both features.

ReferencesCodeFigure would identify the first sentence as code description due to the presence of the word “Fig. 1”, which is the figure number for the code snippet described in this paragraph. We would assign a score of 3 to this sentence. The next sentence contains code identifiers “CredentialCache” and “get”, found in the described code snippet of the research article. So *ContainsCodeIdentifiers* would indicate the second sentence is code description and assign a score of 2. The third sentence would not be identified as seed of a code description by any of our heuristics. The fourth sentence would be identified by *ContainsCodeIdentifiers* since it contains the code identifier “cache”, found in the described code snippet. Hence, this sentence would be assigned a score of 1. The fifth sentence would be identified by *ReferencesCodeByPosition* due to the presence of the phrase “method invocation”, and assigned a score of 2. The next sentence would also be identified by *ContainsCodeIdentifiers* since it contains the code identifiers “get” and “getFullyQualifiedHostName”. However, the same sentence would also get an additional score by *TextBefore* since this sentence is found immediately above the code snippet that it describes. Hence, the last sentence would be assigned a score of 2 by *ContainsCodeIdentifiers*, and add a score of 1 for *TextBefore*, making the total score of this sentence 3. At this point, all the seeds are identified, and the minimum of at least 50% of the total number of sentences in the paragraph to include the whole paragraph would mark the whole paragraph as code description text.

3.4 Enabling Web-scale Code Clone Detection and Search

Our work [3] centers on the combination of three filtering heuristics, two of which have been previously proposed for token-based code clone detection: prefix filtering, token position filtering and adaptive prefix filtering. Prefix filtering was proposed by researchers so that the number of candidate pairs in token based code clone detection can be significantly reduced, improving performance and scalability. As a complement to prefix filtering, token position filtering utilizes the position of the tokens in the code block to further reduce the number of candidate pairs. The third heuristic, adaptive prefix filtering is an extended version of prefix filtering that looks for beneficial opportunities to filter candidate pairs even more aggressively, aimed at even further improving performance and scalability to large datasets. SourcererCC [42] has implemented the first two heuristics, while in our proposed code clone detection technique we have applied all three heuristics.

For a token-based approach of code clone detection, at first, we must convert the source code into a set of tokens. To this end, we extract the set of string literals, keywords, and identifiers in each code block, removing all the special characters, operators and comments. Each of the extracted tokens can occur several times in a code block so we annotate each token with its local occurrence frequency. The sum of the local occurrence frequencies for a specific token across all the code blocks in the corpus is its global occurrence frequency. We sort all the tokens, in each code block, based on their global frequency in ascending order. When there is a tie, the tokens are arranged in alphabetical order.

In token-based code clone detection techniques, two code blocks are classified as clones if they match a certain number of tokens. The required number of matched tokens depends on the user defined threshold value, which ranges between 1 and 10 (i.e. 10% to 100%), and the length of the code blocks. A similarity function (e.g., edit or hamming) measures the degree of similarity between two code blocks. In this project we used the simple overlap similarity function that measures the number of tokens in the intersection between two code blocks.

To determine whether two code blocks, CB_1 and CB_2 , are code clones, we take the maximum value between the number of tokens in each of the code blocks CB_1 and CB_2 and multiply it by the similarity threshold value, θ . A code clone detection technique has to examine at least that number of tokens to determine whether a pair of code blocks are code clones. There is a significant performance cost in performing this token comparison for a large source code repository containing numerous code blocks. The prefix filtering heuristic improves on this by first determining if a strict comparison is necessary, or if the pair of blocks can be filtered out.

According to the prefix filtering heuristic, if the sorted tokens of a pair of code blocks match at least one token in their prefix then these blocks are a code clone candidate pair, which we verify whether are really code clones to each other in a subsequent verification step. On the other hand, if not a single token is matched, then we can discard them from further consideration. The property for prefix filtering is as follows.

Property 1: If two code blocks CB_1 and CB_2 consist of t terms each, which follow an order O , and if $|CB_1 \cap CB_2| \geq i$, then the sub-block CB_{sb1} consisting of the first $t - i + 1$ terms of CB_1 and the sub-block CB_{sb2} consisting of the first $t - i + 1$ terms of CB_2 must match at least one token.

For the second heuristic, token position filtering, we derive an upper bound by summing of the number of current matched tokens and the minimum number of unseen tokens between two code blocks. If this upper bound is smaller than the needed threshold we can safely reject this code block pair. We derive the property for token position filtering as follows.

Property 2: Let blocks CB_1 and CB_2 be ordered and there exists a token t at index i in CB_1 , such that CB_1 is divided into two parts, where $CB_1(\text{first}) = CB_1[1 \dots (i-1)]$ and $CB_1(\text{second}) = CB_1[i \dots (|CB_1|)]$. Now if $|CB_1| \cap |CB_2| \geq \theta * \max(|CB_1|, |CB_2|)$, then for all t in $(CB_1 \cap CB_2)$, $|CB_1(\text{first}) \cap CB_2(\text{first})| + \min(|CB_1(\text{second})|, |CB_2(\text{second})|) \geq \theta * \max(|CB_1|, |CB_2|)$.

Adaptive prefix filtering heuristic is the extended version of prefix filtering, where rather than matching one token in the prefixes of code blocks, we match more, while deepening the size of

the prefixes. This variant defines an ℓ -prefix (instead of a 1-prefix scheme), where ℓ is the number of tokens we want to match. The size of the prefix (i.e., number of tokens within prefix) changes. Adaptive prefix filtering reduces the number of candidates more aggressively, at the cost of more comparisons in the filtering step. An advantageous value of ℓ can be selected based on the cost calculation framework for each of the code blocks. We derive the property for adaptive prefix filtering as follows.

Property 3: If blocks CB_1 and CB_2 consisting of t terms each, which follow an order O , and if $|CB_1 \cap CB_2| \geq i$, then the sub-block CB_{sb1} consisting of the first $t - i + \ell$ terms of CB_1 and the sub-block CB_{sb2} consisting of first $t - i + \ell$ terms of CB_2 will match at least ℓ tokens.

We now describe how a system that uses adaptive prefix filtering can be implemented in practice, which is crucial to making this technique useful. A delta inverted index presents an efficient data structure for clone candidate filtering, while prefix cost calculation is important in determining the size of the prefix ℓ that optimizes the tradeoff between greater reduction in candidate pairs and the added cost of deeper filtering. Both the data structure and cost calculation are crucial steps in implementing the adaptive prefix filtering heuristic in practice.

An inverted index data structure is commonly used to retrieve matching documents (i.e., code blocks) using a particular token as a query (implemented in popular tools such as Apache Lucene [67]). For prefix filtering and token position filtering only a single inverted index data structure is required. Instead of creating an index for each document, an inverted index is created based on each token where it stores all the documents which contain that particular token. That is why it is named as inverted index. However, adaptive prefix filtering requires a separate inverted index for each of the ℓ prefix schemes. A delta inverted index [51] overcomes the repetition that would occur if a simple set of inverted indices were used for adaptive prefix filtering. We describe this data structure.

The requirement for inverted list $I_\ell(e)$ is to store all the code blocks whose ℓ -prefix set contains the token e . Similarly, $I_{\ell+1}(e)$ stores all code blocks whose $\ell+1$ prefix set contains e , and $I_\ell(e)$ is a subset of $I_{\ell+1}(e)$. A delta inverted index $\Delta I_{\ell+1}(e)$ data structure eliminates repetition by only storing the different code blocks between $I_\ell(e)$ and $I_{\ell+1}(e)$. At the outset, inverted index $\Delta I_1(e) = I_1(e)$, and as l increases we create delta inverted indexes $\Delta I_2(e)$, $\Delta I_3(e)$, ..., $\Delta I_t(e)$ for $I_1(e)$, $I_2(e)$, $I_3(e)$, ..., $I_t(e)$ where $(1 \leq \ell \leq t - 1)$.

The delta inverted index up to $\ell = 3$ is shown in Figure 4 for threshold value 0.8. Each of the three large boxes, ΔI_1 , ΔI_2 , and ΔI_3 , represents a delta inverted index that can be queried separately, which contains a set of tokens mapped to their respective containing code blocks. ΔI_2 contains code blocks that are not present in I_1 but present in I_2 . Index ΔI_3 contains code blocks that are not in I_1 and I_2 but are present in I_3 . As an example, for token return inverted index $I_1(return)$ contains CB_1 and inverted index $I_2(return)$ contains CB_1 , CB_2 , and CB_5 , while delta inverted index $\Delta I_2(return)$ contains CB_2 and CB_5 .

In order to select the appropriate prefix length for each of the code blocks, we need to optimize the trade-off between filtering cost (i.e., the cost of looking up tokens deeper in the delta inverted index) and verification cost (i.e., the cost of determining if a pair of code blocks are actual clones

by comparing all of the necessary tokens). The adaptive prefix filtering technique iteratively estimates the cost for the ℓ -prefix scheme and the $(\ell + 1)$ -prefix scheme, and if the $(\ell + 1)$ -prefix scheme's cost is greater than the ℓ -prefix scheme cost then we select the ℓ -prefix scheme for that particular code block. Otherwise we continue to compute the next prefix scheme's cost. Prior results include the fact that this technique selects a global minimum for the optimal prefix scheme [51]. Without cost calculation this technique is not adaptive at all.

ΔI_1								ΔI_2			ΔI_3				
if	static	public	return	for	n	0	5	return	for	n	1	factorial	return		
CB ₁	CB ₁	CB ₁	CB ₁	CB ₂	CB ₂	CB ₃	CB ₄	CB ₂	CB ₄	CB ₃	CB ₁	CB ₂	CB ₃		
CB ₃	CB ₂	CB ₂		CB ₃		CB ₅			CB ₅					CB ₅	
CB ₅	CB ₃	CB ₃													
		CB ₅													
String	args	main	void	else											
CB ₄	CB ₄	CB ₄	CB ₄	CB ₅											

The average cost of verifying a code block CB is $cost_v(CB)$. We compute this cost using s_u and s_l , the upper bound and lower bound of the sizes of all the code blocks within code repository R . We express this via the following equation:

$$cost_v(CB) = |CB| + ((s_{|u|} + s_{|l|}) / 2) \quad (4)$$

To estimate the candidate set size of the ℓ -prefix scheme, $C_1(CB)$, we simply calculate the number of code blocks which appear in at least one inverted list using the tokens within the ℓ -prefix scheme.

Those code blocks that appear in more than ℓ number of inverted lists of tokens within ℓ -prefix scheme, also appear in the candidate set of the $(\ell + 1)$ -prefix scheme. All other code blocks which appear in at least ℓ number of inverted lists of tokens within ℓ -prefix scheme can also appear in the candidate set of the $(\ell + 1)$ -prefix scheme if and only if these code blocks appear in the additional delta inverted lists of the $(\ell + 1)$ -prefix scheme. We take the summation of the size of these two sets where one set contains the intersection of the code blocks appearing both in the ℓ -number of inverted lists in ℓ -prefix scheme and the additional delta inverted lists of the $(\ell + 1)$ -prefix scheme and the other set contains the code blocks appearing in more than ℓ number of inverted lists in the ℓ -prefix scheme.

The adaptive prefix filtering technique can also be utilized for code clone search, where a user-specified code block (i.e., the query) is matched in a corpus consisting of numerous code blocks. In code clone search, different from code clone detection, we do not pre-specify the similarity threshold value before the index is built from the corpus, as we would like for the same index to be able to serve different queries with different threshold values. Therefore, the index structure should be able to deal with any threshold value, $1 \leq |s| \leq 10$, where $|s|$ is the maximum threshold value the index could serve.

As a naive approach, we can build an index for all possible threshold values from 1 to $|s|$ for each code block. However, this would take up huge space and be very time consuming. Instead of building delta inverted indices for each threshold value from 1 to l , we can build delta inverted indices for the maximum threshold value $|s|$, i.e., for 100% similarity (threshold value 10). With this maximum threshold value we build delta inverted indices, ΔI_1 , ΔI_2 , ΔI_3 and so on, until we reach the maximum prefix scheme. At retrieval time, we use this data structure as the means to answer code clone search queries with retrieval-time similarity thresholds. Apart from this modification to the data structure, the algorithm follows the same logic as in code clone detection.

4.0 RESULTS AND DISCUSSION

In this section we report our research results and discuss the implications of those results. As in Section 3.0, this section is organized by research accomplishment. That is, each heading in this section corresponds to the results and discussion for a specific research accomplishment.

4.1 Understanding What Information about Code Snippets Is Available in Different Kinds of Developer Communications

For each document kind in our study of code snippet characteristics, Table 5 shows the number of instances in the study (with a total of 60 document instances), the mean number of code snippets in the unit of analysis for that document kind, the mean number of LOC in those code snippets, and the mean number of lines of natural language text in the unit of analysis for that document kind. Counting the number of code snippets in a document is straightforward except for code reviews.

For code reviews, the code snippets are not embedded in the document; instead, they are being described and discussed and attached to the code review (via hyperlinks). Thus, to count the number of code snippets in a code review, we examined the files that are attached to the review. Each file that is attached is two screens showing a diff. We computed the number of code snippets for a code review by counting each file as two code snippets and computed the number of LOC for each code snippet as the size of the displayed code in the code window, which includes the diff and context LOC.

Counting the number of LOC per code snippet is straightforward; however, counting the number of lines of natural language text as a comparable measure across document kinds is complicated by the fact that some document kinds have different lengths of text lines. For instance, a research paper could be two columns of text, while a blog post might have a different length of lines than mailing lists. Thus, to compute the number of lines of natural language text, we counted the number of characters of natural language text in each document, and divided that number by 80 as an average length of a line. This provides the information in a more traditional unit of number of lines rather than characters, and normalizes the measure over different document kinds.

The mean number of code snippets per unit of analysis of each document kind varies from 1.2 for public chats to 8.6 for blog posts and research papers. Blog posts, code reviews, research papers, and presentations have the higher number of code snippets (> 7), while mailing lists and public chats have only between 1.2 and 2.6 code snippets. Thus, mailing lists and public chats might not prove to be the richest resources for mining code snippets.

The mean length of code snippets varies from 8 LOC in public chats and documentation to 47 LOC in mailing lists. In the case of mailing lists, developers often included entire classes or methods to provide context for their questions. Many of the document kinds contain code snippets in the range of 10 to 13 LOC. This size code snippet may indeed be code examples for single,

specific actions that would be useful to examine further for mining to help developers in different contexts.

As expected, research papers are outliers in size of natural language text in the document with over 400 mean lines of text. However, it is interesting to note that benchmarks, blog posts, and code reviews provide from 65 to 88 mean lines of text, while all the other document kinds range between 12 to 33 lines of text. This may suggest that research papers, benchmarks, blog posts, and code reviews provide more opportunity for mining descriptive information about the code snippets.

In summary, there are wide variations in the numbers of code snippets, mean size of code snippets and mean lines of natural language text across the different document kinds. Larger numbers of available code snippets per document provide more opportunity for code snippet mining. More text per document provides more opportunity for descriptive information about the embedded code snippets. The kinds of information that is available in the text could vary, so we also investigated what kinds of code-related information is available in the document text for each kind of document.

Table 5. Code Snippet and Description Availability by Document Kind

Document Kind	# Documents	Mean # Code Snippets	Mean # LOC per Snippet	Mean # Lines of Text
Benchmarks	2	4.5	13.0	86.7
Blog posts	10	8.6	13.1	88.3
Bug reports	6	2.7	20.1	17.2
Code reviews	7	7.1	33.3	64.9
Course materials	3	3.0	12.8	12.6
Documentation	6	3.5	7.8	22.8
E-books	5	2.6	21.4	37.4
Mailing lists	5	1.6	46.6	17.6
Papers	5	8.6	10.3	439.9
Presentations	3	8.3	11.6	18.7
Public chats	5	1.2	8.3	15.6
Q&A sites	3	4.7	12.6	32.8

Our human annotators and the coding of their observations were used to gain insight into the kinds of information that the natural language text provides about embedded code snippets in different document kinds. Specifically, we focused on identifying whether there are certain kinds of information prevalent across many different document kinds and whether there are kinds of information embedded only in certain kinds of documents. Using the code snippet-related text identified by the human annotators, as well as the observations that they made from that identified text in 31 distinct documents and our labeling of the annotators' observations, we created the heat map in Figure 5. The heatmap illustrates the frequency of occurrence of the different types of information indicated by our eight labels for the code snippets in each document kind. A darker-

colored square indicates a higher frequency of occurrence for a particular type of information in a particular document kind, while a lighter-colored square indicates a lower frequency of occurrence. We created the heat map using mean occurrences of each label across the instances of each document kind. For example, the four mailing list instances that were coded contained, in aggregate, seven code snippet-related observations that were coded as Erroneous, resulting in a mean of $7 / 4 = 1.75$ occurrences of error-related information in mailing lists.

The heatmap in Figure 5 indicates that Explanatory information is prevalent across most of the document kinds that we considered, and is the dominant category for several document kinds, such as blog posts, documentation, e-books, papers, and public chats. This aligns with that fact that the main purpose of these document kinds is to explain aspects of the implementation in the code snippets. Another kind of information that shows up fairly often across different document kinds is design information, which includes programming language, framework, and time/space complexity of the code snippet. Based on this exploratory study, information about origin and clarity of the code snippet is rarely available in any of the studied document kinds. Information about efficiency is found almost exclusively in bug reports, while testing information was found almost exclusively in code reviews.

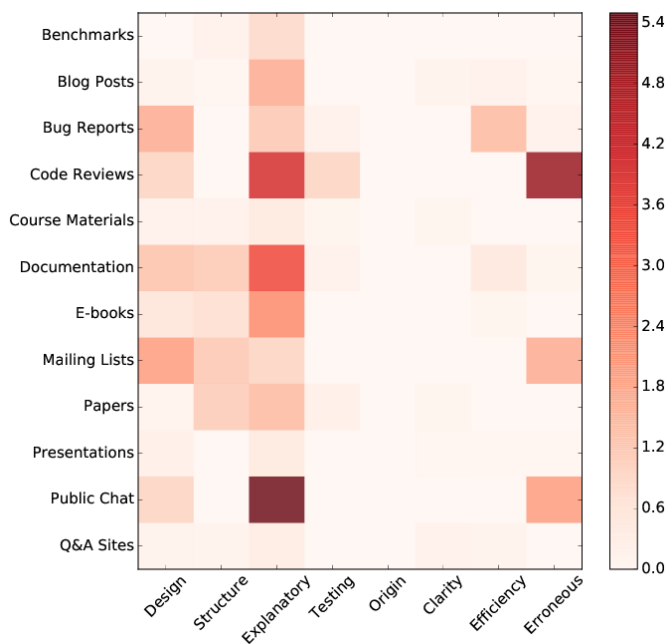


Figure 5. Heatmap of the Frequency of Information Types in Different Document Kinds

By examining the heat map from the perspective of an individual document kind, bug reports, code reviews, documentation, mailing lists, papers and public chats appear to contain the highest amount of diversity in information about code snippets, relative to other document kinds.

The fourth column of Table 6 depicts examples of words and phrases that appear in the document text highlighted by the annotators. Based on these samples, we note that sometimes individual words are adequate cues for a given kind of information (e.g., a name of a programming language or framework), while sometimes a cue requires a phrase, or sequence of words (e.g., “see how much cleaner it is”, “hard to parse”). In fact, for most kinds of information, phrases are needed to cue the kind of information available in the text. These phrases are quite different across information types, which suggests that automating the detection of the kind of information embedded in the natural language text about a code snippet is not trivial. Detection techniques will most likely require natural language processing and machine learning.

Table 6. Description of Labels and Sub-labels with Example Cues in Text

Labels	Sub-Labels	Description	Cue Words/Phrases
Design	Language	Programming language	Java; C# class names
	Framework	Framework used	RxJava; class names
	Complexity	Time/space complexity	$O(n \log n)$
Structure	Data structure	Variable types	cast float as int
	Control flow	Control statements used	for loop; if statement
	Data flow	Data flow chains	TPL data flow usage
	Lines of code	Length of code	line 27 follows
Explanatory	Rationale	Why implemented this way	block tridiagonal systems are solved for each direction
	Functionality	What is implemented	3D Fourier transform
	Methodology	How implemented	with NumCPP
	Output	Results of running code	the end result is
	Similarity	Syntactic/semantic similarities	in a similar way
	Modification	Changes to existing code	allows you to do the same thing with code like
Testing		Test code	depicts such a test method
Origin		Origin of code example	adapted from Stack Overflow
Clarity	High	Clean and understandable	see how much cleaner it is
	Low	Unclear or overly complex	hard to parse
Efficiency	Efficient	Efficient code example	returns very quickly
	Inefficient	Inefficient code example	has the effect of delaying
	Assumptions	Conditions for correctness	only 1d and 2d arrays
Erroneous	Compilation	Code fails to compile	invalid typecast
	Runtime	Runtime errors or exceptions	each error is paired with an output

4.2 Comparing the Efficacy of Two Kinds of Developer Communications in Supporting Software Engineering Tools

Via our evaluation we collected data to answer two research questions:

- Is the same information successfully mined in Q&A forums for supporting mining-based SE tools available and prevalent in chat communities?
- Which characteristics of chat communities may inhibit automatic mining of valuable information?

We display the computed results, including the minimum, median/mean, and maximum, for each of the measures we identified for the first research question in Table 7 and Table 8. We preferred medians but used means in cases where the median value was 0. For instance, the median document length for Clojure was 4 sentences in the `clojurians#clojure` channel, and 9 sentences in Stack Overflow, while the maximum document length was 188 and 111 sentences in these sources, respectively.

Table 7. Findings for First Research Question, Part 1 (Min < Median < Max)

Community	Document Length	Snippet Frequency	Snippet Length
<code>clojurians#clojure</code>	1 < 4 < 188	0 < 1 < 77	1 < 13 < 2,563
<code>clojure</code>	1 < 9 < 111	0 < 7 < 204	1 < 15 < 24,614
<code>elmlang#beginners</code>	1 < 5 < 243	0 < 2 < 255	1 < 11 < 3,336
<code>elmlang#general</code>	1 < 4 < 166	0 < 1 < 125	1 < 11 < 3,833
<code>elm</code>	1 < 9 < 92	0 < 9 < 127	1 < 11 < 10,889
<code>pythondev#help</code>	1 < 4 < 125	0 < 1 < 71	1 < 14 < 3,272
<code>python</code>	1 < 8 < 1,213	0 < 5 < 293	1 < 34 < 36,021
<code>racket#general</code>	1 < 6 < 429	0 < 2 < 2	1 < 11 < 3,690
<code>racket</code>	1 < 9 < 93	0 < 8 < 117	1 < 12 < 11,163

Table 9 lists the results of Mann Whitney U tests comparing our measures for Clojure using the *Slack_{topic}* and *SO_{topic}* datasets. We list results only for Clojure due to space considerations. However, the findings are consistent across the remaining channels/tags in our dataset. That is, the results in the table indicate that we can accept our hypothesis for five of the nine measures (at $p < 0.05$), and the results for the remaining channels/tags in our dataset are consistent. All tests are one-tailed, and the ‘Alternative’ column indicates the direction of the hypothesis. For example, for the Document length measure, our hypothesis is that the median number of sentences in a Slack conversation is less than the median number of sentences in a Stack Overflow post. Similarly, for the Bad code examples measure, our hypothesis is that the ratio of bad code examples to good code examples in a Slack conversation is greater than that same ratio in a Stack Overflow post. For the measures computed with *Slack_{manual}* and *SO_{manual}* the ‘N’ column lists the number of

documents in Slack and Stack Overflow, respectively, and the number of topics for measures using $Slack_{topic}$ and SO_{topic} . The ‘U’ and ‘p’ columns list the U statistic and the p-value, respectively.

Table 8. Findings for the First Research Question, Part 2 (Min < Mean < Max)

Community	Links to		API Mentions		Bad Code Examples	Software-specific Term Frequency
	Gists	Stack Overflow	Snippets	Text		
clojurians #clojure	0 < .03 < 3	0 < .03 < 2	0 < .73 < 3	0 < .87 < 3	0 < .56 < 1	0 < .09 < 1
clojure	0 < .01 < 3	0 < .13 < 6	0 < .43 < 1	0 < .43 < 2	0 < .30 < 1	0 < .11 < 0.32
elmlang #beginners	0 < .04 < 4	0 < .02 < 2	0 < .43 < 1	0 < .57 < 2	0 < .32 < 1	0 < .08 < .75
elmlang #general	0 < .03 < 6	0 < .01 < 2	0 < 1 < 2	0 < 1.4 < 2	0 < .47 < 1	0 < .07 < 1
elm	0 < .04 < 3	0 < .12 < 3	0 < 4 < 11	0 < 1 < 2	0 < .41 < 1	.01 < .10 < .23
pythondev #help	0 < .01 < 2	0 < .07 < 4	0 < .2 < 2	0 < .67 < 2	0 < .36 < 1	0 < .10 < 1
python	0 < .00 < 8	0 < .14 < 30	0 < .2 < 1	0 < .2 < 1	0 < .35 < 1	0 < .12 < .67
racket #general	0 < .06 < 3	0 < .01 < 1	0 < .21 < 1	0 < .86 < 5	0 < .41 < 1	0 < .07 < 1
racket	0 < .00 < 3	0 < .01 < 1	0 < .29 < 1	0 < .29 < 1	0 < .23 < 1	0 < .11 < .30

Table 9. Statistical Results for the First Research Question (Clojure)

Measure	Alt.	N	U	p-value
Document length	<	(20, 20)	0.0	< 0.001
Code snippet frequency	<	(20, 20)	12.0	< 0.001
Code snippet length	<	(20, 20)	197.0	0.47
Gist link frequency	>	(20, 20)	96.0	0.009
Stack Overflow link frequency	>	(20, 20)	82.5	0.08
API mentions in code snippets	<	(15, 7)	61.0	0.76
API mentions in text	<	(15, 7)	66.0	0.86
Bad code examples	>	(15, 7)	70.5	0.10
Software-specific term frequency	<	(20, 20)	57.0	< 0.001

In six of our measures Stack Overflow produced larger values (four out of six with statistical significance at $p < 0.05$): Document length, Code snippet frequency, Code snippet length, API mentions in code snippets, API mentions in text, and Software-specific term frequency. In the

remaining three measures, Bad code examples, Gist link frequency, and StackOverflow link frequency, chats produced stronger values, with Gist link frequency the only measure with statistical significance at $p < 0.05$. While both Slack and Stack Overflow include questions with bad code examples, where a developer provides incomplete or erroneous code examples and asks for help fixing them from the community, our results indicate that this happens more often on Slack.

Table 10 and Table 11 present the results for the second research question. Table 10 shows all of the natural language clues (words, phrases, and emojis) used as accepted answer indicators for each analyzed Slack community. The most prevalent indicator is “Thanks/thank you”, followed by phrases acknowledging the participant’s help such as “okay”, “got it”, and other positive sentiment indicators such as “this worked”, “cool”, and “great”. Accepted answers were also commonly indicated using emojis as listed in the table.

Table 10. Indicators of an Accepted Answer in *Slack_{manual}*

Slack Communities	Unique Indicators of Accepted Answers
clojurians #clojure	good find; Thanks for your help; thanks; thank you; cool; this works Emojis: :taco:, :+1:
elmlang #beginners #general	that what I thought; Ok; okay; kk; Thanks; awesome; Ah, Got it; maybe this is what I am searching for; nice; handy trick; that seems reasonable; makes sense; great; I see, I’ll give it a whirl; thanks for the insight!; I’ll try that; I will go with the first suggestion Emojis: :idea:
pythondev #help	thanks; thank you; Awesome!; why didn’t I try that?; Wow, perfect; okay yeah that makes sense; I understand your idea; ah I see; yeah I thought as much; exactly what I was looking for Emojis: :slightly_smiling_face:
racket #general	thanks; thank you; okay; ok; gotcha; Yep that seems to have worked; Ah fair; that really helps; Happy branch day! Emojis: :tada:

Table 11 presents the quantitative results for the remaining measures. Results represented as percentages are reported directly, while other results, computed as simple counts, are reported as minimum < median < maximum. For example, participant frequency in *Slack_{topic}* ranged from a single participant in a conversation to 26 participants, with a median of 2 participants. A single participant conversation was one with no answers.

Based on the results reported in Table 11, approximately 7% of questions go unanswered, while the maximum number of answers was 4 for a question. Of the questions with answers, approximately 23% of questions have no accepted answer. The last two results indicate that the number of incomplete sentences is only 9% while the noise in a conversation can be as high as

20%. These factors are challenges in mining information automatically from chat communities.

Table 11. Findings for the Second Research Question

Measure	Results
Participant frequency	$1 < 2 < 26$
Questions with no response	6.92%
Answer frequency	$0 < 1 < 4$
Questions with no accepted answer	22.64%
Natural language text context	$1 < 4.5 < 20$
Incomplete sentences	9%
Noise in document	20%

Code snippets have from 1 to 20 sentences that contained some kind of descriptive information about the code snippet. To gain insight into the contextual information provided in the natural language text surrounding code snippets in Slack, we manually analyzed the kinds of information provided by these sentences. Using the labels defined in Table 6, we categorized the information as providing context in terms of design, efficiency, erroneous nature of the code, explanatory description of the functionality of the code, or a detail about the structure of the code.

Figure 6 presents the prevalence of each type of information in our *Slack_{manual}* dataset. We specifically computed the percentage of descriptive sentences of each type. We observed that the most prevalent types of information are “Structure”, “Explanatory” and “Design”. This aligns with the fact that the main purpose of developer chat communication is to explain different aspects of the implementation of the code snippets, such as loop structures, control/data flows, framework and programming language, and time and space complexity.

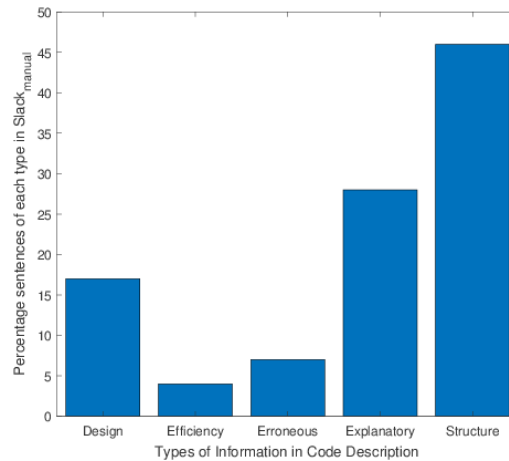


Figure 6. Percentage Sentences of Each Type in $Slack_{manual}$

Next, we discuss the implications of our results on automatic mining of chat communities for use by tools that support software development tasks.

Much of the mining of Stack Overflow for software engineering tools focuses on code snippets, API mentions, and software-specific terms, thus we focus on those results here. For all of these measures, we found that Slack conversations indeed contain all of these items — code snippets, API mentions and software-specific terms. Thus, chat communities provide availability of all of this information for mining. We now look more deeply into how abundant this information is with respect to Q&A forums.

From the results in Table 9, we learn with statistical significance that code snippets occur less frequently in Slack than Stack Overflow. In addition, Table 7 indicates that Stack Overflow posts can have a much larger number of code snippets than Slack conversations. This suggests that code snippets could be mined from chat communities, but more conversations than posts would need to be analyzed to extract similar numbers of code snippets from Slack. We also observed this as we were curating our datasets for this study. Part of the reason for less code snippets within Slack conversations is that participants often include links to Gist instead. Gist links occur more often in Slack conversations than Stack Overflow posts, with statistical significance. This also makes sense as Stack Overflow generally discourages links to external resources, with the exception of other Stack Overflow posts with similar content, which are allowed.

The median code snippet lengths from both sources are similar, although the variation of snippet length is much larger for Stack Overflow code snippets than those included in Slack conversations. Thus, if one wants to mine sources where longer snippets might be available, this data suggests to mine Stack Overflow rather than Slack.

When we focus on code examples in which the natural language text indicates that there is an error or other undesirable property (e.g., inefficient), we see that Slack contains more bad code examples per conversation than the posts on Stack Overflow. This seems intuitive given that Slack

encourages informal less thought-out questions, and provides quicker feedback (and less deliberation) than Stack Overflow. The implication is that Slack may be a better source for mining bad code examples for helping software developers or software tools that learn from bad examples.

While our dataset did not show statistical significance for the difference in API mentions, in either code snippets or natural language text, between conversations and posts, we did observe some evidence that there are less API mentions in conversations in general than posts. Both sources had a fairly low frequency of API mentions. During our manual analysis, we also observed that APIs are mentioned more consistently over all posts than in general Slack conversations. This result suggests that Slack could be used for mining based on API mentions, similar to Stack Overflow, but possibly needing more data for tools looking for more information to augment API documentation, or with less success for traceability of conversations to APIs.

Software-specific terms occurred less frequently in Slack conversations than Stack Overflow posts, with statistical significance. This implies that more conversations than posts would be needed to build thesauri and knowledge graphs using only Slack conversations. This is expected as we expect to find more informal chatter in conversations than the more formal Q&A forum.

Conversations and posts are the most natural unit of granularity for representing a document in our analysis of chats and Q&A forums, respectively. Our data shows that median conversation length is indeed shorter than median post length with statistical significance. Thus, it is possible that some of the reason for lower frequencies of API mentions and software-specific terms is due to conversations being smaller than posts. It could also be affected by the use of links to Gist and Stack Overflow in conversations, instead of natural language text.

At first sight, one might believe that the informal nature and interleaved conversations of Slack would make it difficult to mine automatically. There is also no pre-defined notion of conversation; each could span from two messages to hundreds. However, our work to implement this study required us to disentangle conversations. We found that adapting the technique and training sets can indeed achieve high accuracy in disentangling the Slack conversations.

Moreover, Slack conversations are only available to people who join the community channel, and free Slack teams have a 10,000 post limit, which also caps the amount of data history for analysis. To mitigate this challenge, we developed scripts using Slack API to automatically download conversations on a regular basis, and we plan to collate them over time to have a significant data size for analysis.

Based on Table 11, the number of questions with no accepted answer is similar to Stack Overflow. Thus, there are accepted answers available in Slack. The median number of answers is only one answer. This suggests that there are expert people on the channels, such that only one answer is given and it is accepted. One might think it is then easy to automatically identify the answer to a given question, especially with the use of emojis and the word/phrase indicators of accepted answers. However, an automatic mining tool needs to automatically identify the sentence in a conversation that is an answer to a question and which question it is answering. This implies that natural language processing techniques and sentiment analysis will most likely be needed to automatically identify and match answers with questions.

Table 11 results of a median of 5 and up to 20 sentences pertaining to a code snippet suggest that there is plenty to mine about code snippets. Figure 6 shows that most of that text is describing what the code does, something about its structure or its design. All of this information would be quite valuable along with the code snippet, if it could be extracted automatically. From our manual analysis, we believe that some of this information could be mined with advanced natural language processing and machine learning approaches. The 20% noise in conversations further complicates the problem. Incomplete sentences can make it difficult for classic natural language processing (NLP) techniques, but we only observed 9% sentences in conversations as incomplete. A key challenge is how to develop techniques that do not require tedious, large amounts of human annotations for training.

4.3 Extracting Code Snippets and Their Descriptions from Research Articles

Via our evaluation we collected data to answer the following research questions:

- How effective is our approach to automatically identify code descriptions in natural language text of research articles?
- What kinds of information are available in natural language text describing code snippets in research articles?
- How do authors typically reference code snippets within their code description text in research articles (i.e., what cues are most prominent?)

Our code description identification process is fully automatic. It takes XML with markup classification of code and natural language text of a single article as input and outputs XML with additional markup for code description text, as well as data for our evaluation study. Due to the inaccuracies of the tools for pdf-to-text conversion and optical character recognition (OCR)-to-text conversion that we experienced, the preprocessing is currently semi-automatic. That is, we apply current state of the art tools to convert to text, but then manually clean up the inaccuracies, so that our evaluation study is not affected by the inaccuracies of the preprocessing.

The subjects in our study are research articles (disjoint from our development set) that contain in total 100 code snippets, selected from ACM DL and IEEE Xplore in the domain of software engineering. Because many articles have more than one code snippet, our final evaluation set consists of 4 journal papers and 4 conference papers published between 2011 and 2015.

To answer the first research question, we measured the effectiveness of the overall precision and recall of the code description identification, and also the precision of the seed identification. We did not compute recall of the seed identification because we did not want to reveal details of our approach to the human annotators in creation of the gold set. Precision is calculated by determining the percentage of automatically identified code description sentences that are marked as code-related descriptions by human judges. Precision of the seed heuristics is computed similarly, instead focusing on only those automatically identified seed sentences. Recall the overall code description identification is computed by determining the percentage of all the sentences that

describe the code snippets in the study (as identified by human judges) that are also identified as code descriptions by the automatic technique.

To answer the second research question, we computed the frequency that each seed heuristic was triggered, including counts for each time a given heuristic is triggered more than once on a given sentence.

To answer the third research question, one of the authors used the results from the work described in Section 4.1 and manual analysis of the human annotated sentences to develop a labeling scheme to code the annotated sentences. We defined six major categories of labels, or codes, and 20 sub-labels for the observed code properties as described in Table 6. The third research question is addressed by coding each annotated sentence and computing the frequency of occurrence of each label in the subject set of research articles.

We created a gold set for our evaluation by recruiting human annotators. Our human annotators consisted of 10 computer science students --- 9 graduate students and 1 senior undergraduate researcher. These participants had no knowledge of our techniques, are not authors of this report, and are equipped with prior computer science and programming experience.

We designed a set of instructions and had two of the participants test the annotation procedure while keeping a note of the time they required for each code snippet. Based on the timing results, each of the ten judges was assigned research papers for 20-30 of the randomly selected code snippets. To account for potential subjectivity of human opinion, each of the 100 code snippets was analyzed by two judges separately. Therefore, in total, we collected 200 annotated objects for this evaluation study. Since there were inconsistencies in some of the human annotations, we considered any sentence that either annotator highlighted in our evaluation as a code description.

Specifically, the judges were instructed to annotate natural language text in the papers with the following instructions: “Your task is to review several assigned research papers and highlight any text in the entire paper that you think is describing an embedded code snippet or any property of the code snippet (highlighted in yellow in the document), and label each highlighted text with the related code snippet number”.

For our evaluation data set, the humans annotated 745 sentences as code descriptions. The gold set does not include any captions. We did not ask the human judges to highlight the captions of the figures containing code snippets in the evaluation set, since we assume that a caption to a figure containing code is always relevant to that code, and we did not want the captions to bias the results.

As part of evaluating the effectiveness, we considered several configurations for code description identification: (1) Should all seed heuristics be treated equally or with different scores reflecting their perceived accuracy? (i.e., What scoring approach provides better precision?) (2) For each of the seed heuristic scoring schemes, which threshold provides higher precision? (3) How does the minimum number of seed sentences used to identify neighboring code-related text affect the precision and recall of CoDesNPub Miner? Note that we were most interested in higher precision than recall because we want the identified descriptions to indeed be descriptive, whereas missing some descriptions is not critical.

Table 12 presents results to answer the first two research sub-questions, by reporting the precision for the two seed scoring schemes under three thresholds. The precision is the same for threshold of 1 because it indicates that only one heuristic is needed to identify a seed, in either scoring scheme. Higher thresholds with equal scores mean at least 2 or 3 heuristics, respectively, need to indicate a seed. In the equal scoring scheme, requiring two heuristics for a seed identification provides the highest precision, in fact, higher than any of the thresholds with the accuracy-based schemes.

Table 12. Precision of Seed Heuristics

Scoring Scheme	Thresholds		
	1	2	3
Equal score (= 1)	62.69	80.26	71.42
Accuracy-based score	62.69	69.33	72.89

Table 13 addresses the last research sub-question by reporting precision and recall for different neighboring code-related text identification using the best scoring and threshold combination for identifying seeds. Table 13 shows results for four minimum number of seed sentences needed to consider the whole paragraph as code description text. As expected, the precision is higher at the higher minimums ($\geq 50\%$ and $\geq 75\%$), with a tradeoff of reduced recall. With higher precision in the identification as a priority over missing descriptions, the higher minimums would be used.

Table 13. Effectiveness of Code Description Identification

Minimum # of Seeds	Precision	Recall
$\leq 24\%$	39.05	70.20
$\geq 25\%$	53.41	50.33
$\geq 50\%$	66.04	28.45
$\geq 75\%$	68.30	20.53

Our qualitative analysis focuses on: *When our system is not effective, what is the breakdown between, and the characteristics of, incorrectly identified code descriptions and missed code descriptions?* We examined the evaluation set where our best configuration either missed code descriptions or incorrectly identified code descriptions. There exist 71 out of 224 sentences (31%) that were identified incorrectly as code descriptions, and 592 out of 816 sentences (72%) that the human judges indicated described code examples, but the system missed them. Table 14 shows

examples from each of these categories along with some correctly identified code description sentences.

Analysis of the sentences incorrectly identified as code descriptions indicates that these sentences were either describing an algorithm (or pseudo code) or referring to figures with statistical analysis from experiments in the article. In the third example in Table 14, the author is explaining the results of an experiment using figures containing charts. This sentence is identified using our seed heuristic *CodeFigureListing*. Our tool is currently not able to discard a sentence that describes figures containing statistical analysis such as tables or charts. In the fourth example, the author explains the intuition behind implementing a functionality. Although this sentence does not describe a code snippet specifically, it gives us some information about the implementation, which might be useful for building code recommendation systems.

Our analysis of the sentences where CoDesNPub Miner missed sentences describing code snippets revealed some limitations of using a system based only on features of phrases contained in sentences. The fifth example in Table 14 contains assumptions of specific code implementation, explaining the pre-conditions needed before implementing an algorithmic step in the code. Absence of phrases indicating explicit mention of code implementation accounted for the tool missing to identify such sentences. Lastly, in the sixth example, the figure referred to in this sentence does not contain real code examples, but rules for an implementation. This sentence contains information about the rationale for implementing a code, which our tool fails to identify, again due to absence of code specific phrases in the sentence.

Table 14. Examples of Analyzed Code Description Sentences

Identified Correctly as Code Descriptions
First, we notice that EVOSUITE uses the method toString rather than getElementName in the assertion.
Listing 9 shows an example of three statements that were single statement blocks after the first phases, but can be merged into a single block because they have similar RHSs.
Identified Incorrectly as Code Descriptions
The results of our initial study are summarized in the form of boxplots in Figure 2, and detailed statistical analysis is presented in Table (a) for Option, Table III (b) for Rational, and finally Table III (c) for DocType.
Since our choice of a particular algorithm may not match what the user needs, having the ability to add user-defined functions was important.
Missed Code Descriptions
Meanwhile, if it appears in a requires clause (i.e., the precondition of the updated version), E should be evaluated in the pre-state of the previous version (i.e., (o1, h1)).
Such a difference is captured in the two topmost rules in Figure 5 (c) where notations “ensures” and “requires” designate the clause in which a prev expression appears.

Figure 7 and Figure 8 depict the frequency distribution of the kinds of information described in the natural language text that was annotated in our gold set, as coded by our labeling scheme and sub-labeling, respectively. Figure 8 indicates that Methodology information is the most prevalent kind of information, which shows that the main purpose of mining code examples with descriptions from articles can be to explain the aspects of their implementation. The second most prevalent kind of information is Rationale, which shows that authors also explain why a code snippet is implemented in a particular way, which could be valuable meta-data for a mined code example for learning.

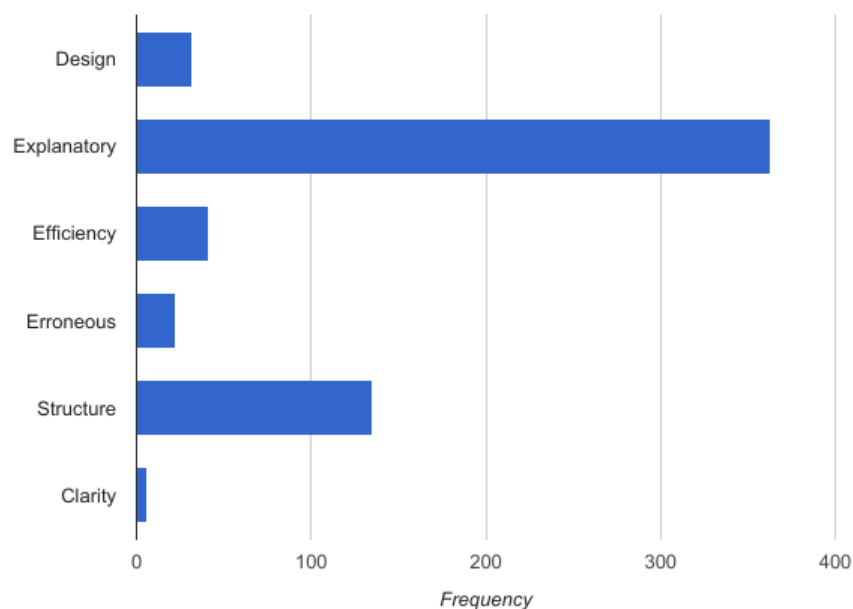


Figure 7. Frequency of Information in Research Articles by Label

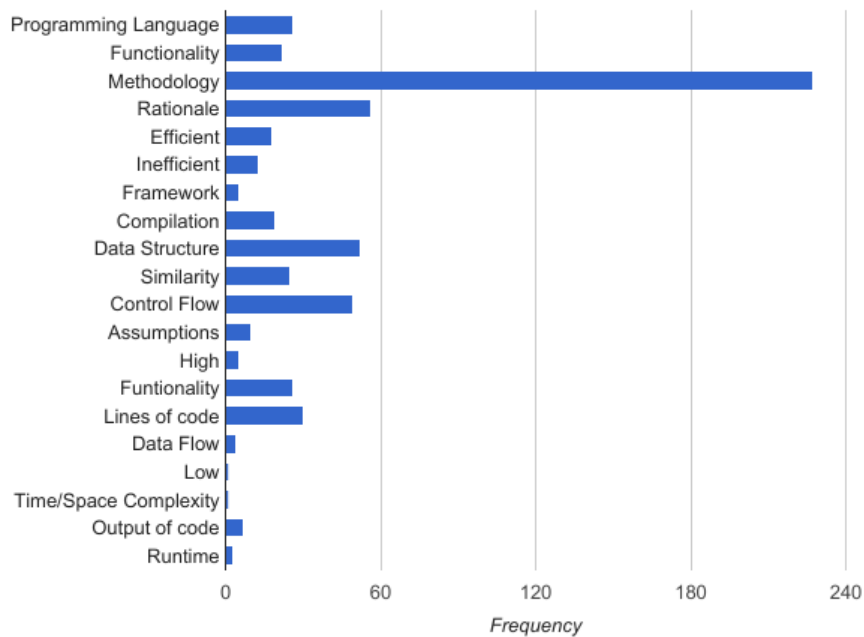


Figure 8. Frequency of Information in Research Articles by Sub-label

These results also suggest that research articles rarely contain overly complex code examples, since they mostly describe novel ways to address a problem rather than going into the details of code complexity. Looking beyond these two categories, we see that a wide variety of information can be gained from descriptions associated with code snippets in digital libraries.

The relative frequency of each feature used to indicate code description text is shown in Figure 9, which depicts that *ReferencesCodeFigure* is the most prevalent heuristic. The next prevalent heuristic is *TextBefore/TextAfter* which helps in identifying sentences that describe less obvious details about code snippets.

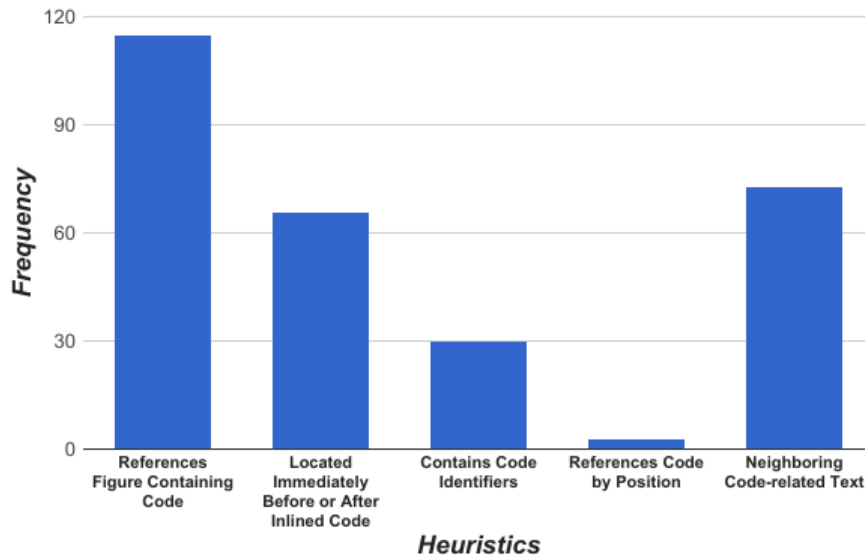


Figure 9. Relative Frequency of Each Feature Indicating Code Description Text

Our subjects are selected from both journal and conference papers in software engineering, across different years from the ACM and the IEEE Xplore digital libraries, which contain millions of full-text documents of publications. The results may not transfer to papers from different disciplines in computer science; we chose publications in the field of software engineering as we believe these contain a large number of analyzable code snippets.

The technique we used to identify code snippets in unstructured documents is capable of identifying code snippets in different programming languages from documents containing code snippets and natural language. All of our heuristics for extracting code descriptions are also programming language independent. Our heuristic *ReferencesCodeByPosition* uses a manually created dictionary of words implying description of code snippets. To create this dictionary, we have selected papers in our evaluation set that contain code examples in various programming languages such as Java, C++, C, and Python.

Research papers often interleave pseudocode and code fragments; however, CoDesNPub miner is not able to distinguish between pseudocode and code fragments. It identifies all code fragments in a research paper, and also cannot differentiate between novel code contributions and code snippets that are used only as examples in an empirical study. Our datasets for development and evaluation consist of papers that do not contain pseudocode.

As with any study based on human annotators for establishing the ground truth, there might be some cases where the humans may not have correctly annotated the descriptions for the code snippets. To limit this threat, we ensured that the human judges had considerable programming experience and research paper reading experiences, and we also ensured that each code snippet

was judged by at least two judges, and when they disagreed, we considered any sentence that either annotator highlighted in our evaluation.

The dataset used for evaluating CoDesNPub consisted of a total of 8 papers including both journal and conference publications. Considering the amount of research work produced in IEEE and ACM publications for the period of 4 years (2011-2015), it is possible that scaling to more than 100 code snippets in our evaluation set might lead to different results. However, we needed to make the human annotation work reasonable to recruit judges.

4.4 Enabling Web-scale Code Clone Detection and Search

Our goal is to implement a code clone detection tool that can scale to massive inter- or intra-project source code repositories and overcome limitations in many existing tools, such as unsustainable execution time, inadequate system memory, restrictions in inner data structures, and exhibiting errors due to their design not expecting a large input [42], [43], [68]. In evaluating our tool, we focused on answering the following research questions.

- *Does adaptive prefix filtering achieve better performance at scale than the best state of the art tool SourcererCC?* SourcererCC is a recent code clone detection tool aimed at scalability on a single machine [42]. Our adaptive prefix filtering heuristic extends the filtering heuristics used by SourcererCC, so a comparison between the two is both natural and necessary. Several code clone detection tools have been benchmarked in recent papers [68], [69]. Among all of the measured tools, four publicly available tools achieve exceptional scalability and accuracy in code clone detection: CCFinderX [45], Deckard [50], iClones [71] and NiCad [72]. In turn, SourcererCC has been measured to outperform these four publicly available and popular tools [42]. Recently, several popular code clone detection tools were compared [73], including VUDDY [73], SourcererCC, CCFinderX and Deckard. SourcererCC outperformed CCFinderX and Deckard. VUDDY outperformed SourcererCC, however, it only detects Type-1 and Type-2 clones.
- *Does adaptive prefix filtering achieve reasonable accuracy in clone detection?* This research question aims to determine whether the adaptive prefix filtering heuristic can achieve reasonable precision and recall, and whether it can serve as a replacement to SourcererCC.
- *Does the application of adaptive prefix filtering to code clone search achieve acceptable query response time?* Searching for similar code fragments in a very large scale source code repository within a reasonable amount of time is a challenging problem. In using our technique for code clone search, we wanted to determine whether this application produces reasonable response time, which is key for it to be useful in practice.

We first describe a comparison of the scalability of the adaptive prefix filtering technique relative to SourcererCC. To answer this research question we relied on publicly available large-scale evaluation datasets, which have recently become available. The IJaDataset 2.0 [54] is a large inter-project Java repository containing 25,000 open source projects with 3 million source files and 250

M LOC. It is mined from SourceForge and Google Code [42]. To compare SourcererCC with adaptive prefix filtering, both tools were benchmarked on a standard workstation with 3.50GHz quad-core i5 CPU, 32.0 GB of memory, and 64-bit Windows operating system. The execution of both tools was scripted to measure the run-time five times for each input and report the average. For adaptive prefix filtering, we used the 2-prefix scheme in constructing the delta inverted index. In order to measure the performance at different input sizes, we blindly selected a subset of the Java source files in IJADataset ranging from 10,000 to 160,000 files, approximately from 1 M to 17 M LOC, such that the selected files at smaller input sizes were also contained in the larger inputs. This ensures a stable measurement since code clone technique execution time may be reliant on clone density [42]. Both techniques consumed extreme amounts of time (>10 hours) at 160,000 files (approximately 17 M LOC), and therefore we did not attempt using larger input sizes. And for the same reason we limited similarity degree from 70% to 90% in comparison. But for the smaller dataset (1 M LOC) we have experimented on all similarity degrees (from 10% to 100%).

Figure 10 shows a comparison in execution time for the adaptive prefix filtering technique and SourcererCC for threshold values 7, 8, and 9 (i.e., 70% to 90% similarity). For threshold values 7 and 8 the adaptive prefix filtering is showing an improvement relative to SourcererCC, but not at threshold value 9. As the input size increases to 160,000 files, the difference in execution time between SourcererCC and the adaptive prefix filtering becomes wider for threshold values 7 and 8.

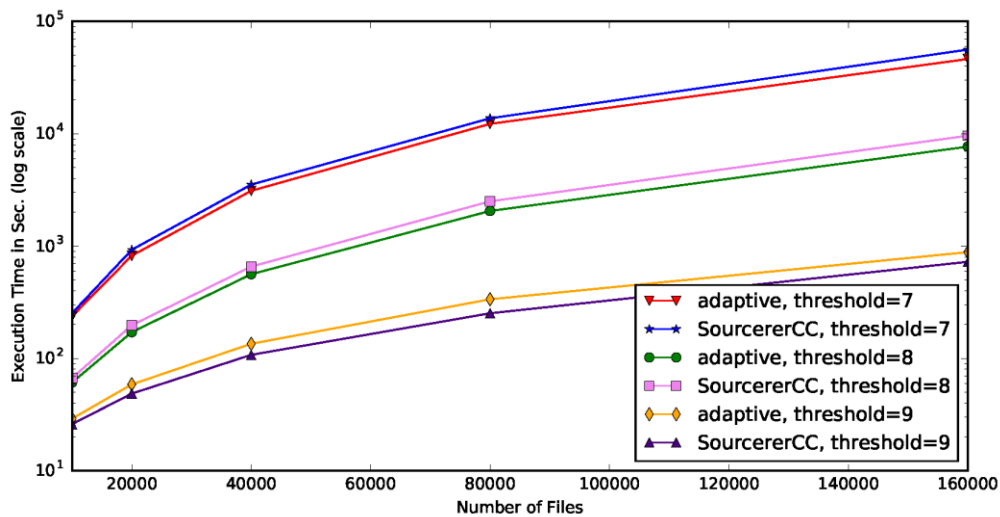


Figure 10. Comparison of Execution Time for Different Input Sizes

In Table 15 we show the execution times for SourcererCC and the adaptive prefix filtering technique for a fixed input of 10,000 files (approximately 1 M LOC), but at varying threshold values. We show both raw execution times as well as the percentage improvement of adaptive prefix filtering technique in the rightmost column. Adaptive prefix filtering performs better than SourcererCC for threshold values ranging from 4 to 8. For threshold value 1, 2, 3, 9 and 10 the execution time of adaptive prefix filtering technique is larger. We argue that the similarity degrees from 40% to 80% are more commonly used. Exact or near-exact similarity (i.e. 100% and 90%) do not make sense for detecting Type-2 and Type-3 code clones, while code clones detected with 10% to 30% similarity are likely to contain a very large number of false positives.

To better understand the effect of the more aggressive filtering performed by adaptive prefix filtering, and understand the rationale behind the performance numbers in Table 15, we compared the number of candidate pairs (in log scale) of SourcererCC and the adaptive prefix filtering technique in Figure 11 for 10,000 files. The candidate pairs are the number of remaining code clone candidates after the filtering that both techniques perform. A reduction in the number of candidates translates to improvement in execution time, after the penalty for the more sophisticated indexing structure and cost calculation of adaptive prefix filtering is factored in. We observe the strongest reduction in the percentage of candidate pairs in the similar thresholds of 4 to 9 where we observed improvements in execution time. Threshold value 9 has a large percentage reduction in candidate numbers, but the actual reduction numbers are very low and insufficient for offsetting the penalty of more aggressive filtering. For threshold value 10 (100% similarity) the percentage decrease is 0. This is logical because with complete similarity adaptive prefix filtering reduces to regular prefix filtering as there is no room to extract a deeper prefix.

Table 15. Comparison of Execution Time in Seconds (10,000 Files)

Threshold Value	Adaptive Prefix Filtering	SourcererCC	% Improvement
1	4,982.23	4,452.31	-10.64%
2	5,357.54	5,339.58	-0.34%
3	4,499.14	4,478.93	-0.45%
4	3,001.19	3,015.76	0.49%
5	1,627.95	1,652.18	1.49%
6	712.45	740.16	3.89%
7	222.68	249.27	11.94%
8	58.84	64.25	9.19%
9	27.25	23.99	-11.96%
10	23.25	21.89	-5.85%

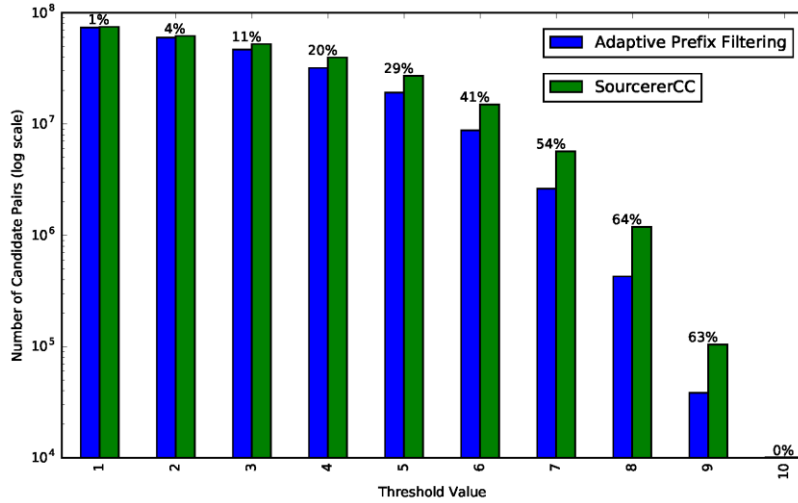


Figure 11. Comparison of Number of Candidate Pairs (10,000 Files)

Fitting with the limited memory budget available on commodity machines is an important factor in scalable code clone detection. For our adaptive prefix filtering technique we use a deeper prefix scheme, resulting in a larger index that will clearly consume more memory than SourcererCC. We measured the memory requirement for both SourcererCC and adaptive prefix filtering for 1 M LOC in the indexing and clone detection phases. While varying the threshold value, we measured how much memory is consumed in each iteration of the indexing process and recorded the maximal amount of memory consumed. The resulting memory requirement was 103 MB for threshold value 1 and 60 MB for threshold value 10, which is 60% and 20% higher relative to SourcererCC's maximal memory footprint in indexing. The memory usage is smaller for the larger threshold value because as the threshold value is increasing the prefix length is decreasing, so fewer tokens need to be stored in the index structure. In the subsequent clone detection phase we measured the memory requirement for each query block, again, selecting the maximum amount of memory required. We obtained 316 MB for adaptive prefix filtering and 204 MB for SourcererCC. In conclusion, although adaptive prefix filtering technique requires higher amount of memory compared to SourcererCC, the memory requirement of both techniques is relatively small and would not impact most deployments. Part of the reason for the small footprint is the efficient allocation of memory by Apache Lucene, which was used to store the inverted index by both SourcererCC and our implementation.

We use precision and recall for measuring the accuracy of the adaptive prefix filtering technique as these are the two most commonly used metrics to determine the quality of a code clone detection technique [74]. Measurement of clone recall and precision has been greatly aided by recent datasets and frameworks like BigCloneEval [75]. This framework can be used for the evaluation of code clone detection tools based on the BigCloneBench clone detection benchmark [76]. BigCloneBench contains a large set of known clones from the inter-project software repository

IJaDataset 2.0, which we used to answer the first research question. Note that SourcererCC and adaptive prefix filtering produce the same clones as output due to the inherent similarity in the techniques.

For measuring recall of adaptive prefix filtering we use file level granularity, which means the clone pairs are actually a pair of two Java source files that are detected as code clones to each other. The evaluation of SourcererCC was at the method level [42]. In measuring recall using BigCloneBench, Type-3 and Type-4 code clones are separated in four categories because it is difficult to separate Type-3 and Type-4 since there is no consensus on the smallest similarity of Type-3 [48]. These four categories are: Very Strongly Type-3 (VST3) clones that have range of syntactical similarity from 90% to 100%, Strongly Type-3 (ST3) clones that have range of syntactical similarity from 70% to 90%, Moderately Type-3 (MT3) clones that have range of syntactical similarity from 50% to 70% and Weakly Type-3 (WT3/T4) clones that have range of syntactical similarity from 0% to 50%, which are often Type-4 code clones [42].

For this evaluation we used a similarity threshold of 70% in executing adaptive prefix filtering, which is the default setting for BigCloneEval. Our technique produced very high recall for Type-1 code clones (97%), and detected Type-2 code clones with a reasonable recall of 77%. For Type-3 clones, the recall decreases significantly, from 60% for the VST3 category, 26% for ST3, 16% for MT3 and less than 1% for WT3/T4. The Weakly Type-3/Type-4 clones have low syntactic similarity which makes it very hard for our technique to detect at the 70% threshold, so this result is not unexpected.

The adaptive prefix filtering technique detects the same code clones as SourcererCC, and therefore its precision (and recall) will be the same as SourcererCC. SourcererCC's precision was previously evaluated via a set of 390 clone pairs, which were manually identified by several researchers with high mutual inter-agreement. Out of these 390 clone pairs, 355 were true positives while 35 were found false positives, resulting in a precision of 91% computed at method level granularity [42].

The value of the precision metric, unlike recall, is influenced by the number of false positives. For token based code clone detection techniques, a common source of false positive clones stems from the fact that these techniques commonly treat the input as a bag of words, ignoring the ordering of tokens in the input. To illustrate this point we consider a false positive clone pair identified by our technique (and likely most other token based techniques). The pair of code blocks constituting the false positive example are: (1) a code block that implements password validation and (2) a code block that implements password encryption. Although these two code fragments have 70% similar tokens they are functionally dissimilar. In fact, the pair of code blocks have significant differences at the line level and their different purpose would be easily observed by a human. However, both of the code fragments are dealing with passwords and there are a significant number of similar tokens. Some of these are: password, passwordInDb, MessageDigest, and update.

We showed that the adaptive prefix filtering technique can easily be extended so that it can support similarity search. Next, we examined the practicality of code clone search based on this technique by measuring the performance of this variant. Our goal here was to show that the use of our technique for code clone search is practical, but not to show that our technique outperforms those

that specialize solely on the code clone search problem. Table 16 shows the time it takes to construct the index, and, more importantly, the average query time for 1,000 files randomly selected from IJADataset with similarity degree 80% (i.e., threshold value 8). By index time, we refer to the time required to build the special indexing structure given a maximum similarity threshold of 100%. After creating this index, we use a random selection of 1,000 Java source files, from the files used to construct the index, performing a code clone search for each file. We report the average search time for different input sizes. In examining the performance of our technique in Table 16, we observe sub-second response times for all queries, even at the larger corpus sizes. This is likely to be reasonable performance for many applications.

Table 16. Performance of Code Clone Search

Number of Files	Similarity	Index Time (s)	Query Time (s)
10,000	80%	70.52	0.046
20,000	80%	160.12	0.095
40,000	80%	351.29	0.177
80,000	80%	736.93	0.368

Our experimental evaluation is limited only to the Java programming language via the IJaDataset. A threat to validity is that our approach may not be effective for other programming languages or other corpora. As a mitigating factor to this threat, we argue that researchers have observed that programming languages possess common statistical traits in their term distribution relative to natural language [77]. Therefore, our technique is likely to exploit these similarity notions and be beneficial across different languages and code bases.

We evaluated adaptive prefix filtering for code clone detection on a commodity workstation, and not on specialized hardware. Our implementation is also multithreaded, which improves performances on modern processors. It is unclear whether the performance improvements we observed translate to other hardware architectures. We also used a fixed time budget of roughly one day in executing our tool, which limited our largest input to 160,000 Java source files containing approximately 17 M LOC. Another threat to validity is that we did not experiment with larger inputs, for which it is possible that our technique fails to perform well. As a mitigating factor, we note that our results span numerous (smaller) input sizes and the trend we observed is towards improved rather than degrading performance compared to SourcererCC as the input size increases.

5.0 CONCLUSIONS

In this section we discuss our conclusions. As in previous sections, this section is organized by research accomplishment. That is, each heading in this section corresponds to the conclusions for a specific research accomplishment.

5.1 Understanding What Information about Code Snippets Is Available in Different Kinds of Developer Communications

For this research accomplishment, we described the exploratory study in which we investigated the types of information available in different kinds of developer communications. Our results address research questions about the characteristics of the code snippets embedded in different kinds of developer communications, the types of information contained in those code snippets, and the cues that indicate code snippet related information. Another contribution of this work is to provide a methodology for conducting such a study, as well as indications of which developer communications are the most useful for mining descriptions or properties of code snippets at a large scale.

5.2 Comparing the Efficacy of Two Kinds of Developer Communications in Supporting Software Engineering Tools

We reported on our exploratory study to investigate the potential of developer chat conversations in Slack as a mining resource for software engineering tools. We found that chat conversations provide much the same information as can be found in Q&A posts on Stack Overflow. However, chat conversations generally provide less of this information (e.g., code snippets, API mentions, and software-specific terms) than do Q&A posts. Yet, chat conversations provide more links to external information (e.g., Gists) and more bad/negative code examples, both of which could be useful for mining-based software engineering tools. We also found that questions posted to Slack are about as likely as questions posted to Stack Overflow to receive either no response or to have an accepted answer. Specifically, about 7% of questions posted to Slack receive no response, whereas about 5% of Stack Overflow posts have no answer. Similarly, about 23% of Slack questions have no accepted answer, compared to 18% on Stack Overflow.

We were able to apply an existing chat disentanglement approach to Slack conversations with minimal adaptation. Further, we found that only 9% of sentences are not (grammatically) complete. Together, these findings suggest that it is feasible to apply automated mining approaches to chat conversations from Slack. However, the lack of inbuilt formal Q&A mechanisms on Slack does result in some potential mining challenges. Notably, identifying an accepted answer is non-trivial and requires both identifying the responsive sentence(s) and the question to which those sentences are responsive. Part of this challenge stems from the free-form style of chat conversations, in which a question may be followed by a series of clarification or follow-up

questions, and contextual clues must be followed to determine which question is ultimately being answered.

Our study reveals several new opportunities in mining chat conversations. As the number of links to external resources in Slack was substantial in our study, and because of its easy integration to many different sources (e.g., GitHub), public Slack channels serve as a clearinghouse for other information available on the Web. Embedded links on Slack are used in many different contexts, and as such can be mined to provide more context to other data sources or even as comments on the quality of other data sources (e.g., by indicating that a Stack Overflow post failed to help a developer).

During our study, we also observed that developers use Slack to share opinions on best practices, APIs, or tools (e.g., API X has better design or usability than API Y). Stack Overflow explicitly forbids the use of opinions on its site. However, it is clear that receiving opinions is valuable to software developers. The availability of this information in chat may lead to new mining opportunities for software tools.

5.3 Extracting Code Snippets and Their Descriptions from Research Articles

This research accomplishment takes the first step toward unleashing the potential to mine the vast number of computer science articles in digital libraries for code snippets that come with useful descriptive information about their functionality and properties. We presented the first technique to automatically identify natural language descriptions of code snippets embedded within articles, where code snippets can be separated as figures that are not located next to their descriptive text. We also evaluated the technique. Our evaluation results indicate that we can achieve precision of 68.30% with recall of 20.53% with a single configuration of scoring and threshold scheme, which is promising. Analysis of the information available in the descriptions shows that a variety of information about code snippets could be learned.

5.4 Enabling Web-scale Code Clone Detection and Search

This research accomplishment comprises a novel code clone detection technique utilizing the adaptive prefix filtering heuristic and its evaluation. Our proposed technique can outperform the most recent scalable code clone detection tool, SourcererCC, in terms of execution time within a certain range of similarity thresholds (from 40% to 80%). Since the technique directly extends SourcererCC, it produces the same output, duplicating the same high precision and recall as SourcererCC. We have evaluated our proposed code clone detection tool by randomly creating a subset of the IJaDataset 2.0, a large code clone benchmark that contains 250 M LOC and 25,000 open source Java systems.

Our experimental results indicate that our adaptive prefix filtering based code clone detection technique can be practically utilized in various code clone detection related applications that

require a large source code repository to be processed on a single machine. In our experiments we successfully performed code clone detection on a Java code base with 17 M LOC within a reasonable time window of several hours.

Our approach is also among few code clone detection techniques in the literature that can be additionally used for code clone search, which is the related problem of retrieving similar code blocks to a single code block issued as a query. Typically, code clone detection techniques make design decisions that allow them to only operate in batch mode, while code clone search requires the flexibility to answer numerous clone queries using a pre-built index. We show acceptable indexing and querying times for this application of our technique.

6.0 REFERENCES

1. Chatterjee, P., Nishi, M.A., Damevski, K., Augustine, V., Pollock, L., and Kraft, N., “What Information about Code Snippets Is Available in Different Software-Related Documents? An Exploratory Study,” *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, Klagenfurt, Austria, 2017.
2. Chatterjee, P., Gause, B., Hedinger, H., and Pollock, L., “Extracting Code Segments and Their Descriptions from Research Articles,” *Proceedings of the 14th International Conference on Mining Software Repositories*, Buenos Aires, Argentina, 2017.
3. Nishi, M.A., and Damevski, K., “Scalable Code Clone Detection and Search based on Adaptive Prefix Filtering,” *Journal of Systems and Software*, 137, 2017, pp.130-142.
4. Bacchelli, A., D’Ambros, M., and Lanza, M., “Extracting source code from e-mails,” *Proceedings of the 18th International Conference on Program Comprehension*, 2010, pp. 24–33.
5. Panichella, S., Aponte, J., Di Penta, M., Marcus, A., and Canfora, G., “Mining source code descriptions from developer communications,” *Proceedings of the International Conference on Program Comprehension*, Passau, Germany, 2012, pp. 63–72.
6. Vassallo, C., Panichella, S., Di Penta, M., and Canfora, G., “CODES: Mining source code descriptions from developers discussions,” *Proceedings of the 22nd International Conference on Program Comprehension*, Hyderabad, India, 2014, pp. 106–109.
7. Treude, C. and Robillard, M., “Augmenting API documentation with insights from Stack Overflow,” *Proceedings of the 38th International Conference on Software Engineering*, Austin, Texas, USA, 2016, pp. 392–403.
8. Petrosyan, G., Robillard, M., and De Mori, R., “Discovering information explaining API types using text classification,” *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 869–879.
9. Wong, E., Yang, J., and Tan, L., “AutoComment: Mining question and answer sites for automatic comment generation,” *Proceedings of the 28th International Conference on Automated Software Engineering*, 2013, pp. 562–567.
10. Amintabar, V., Heydarnoori, A., and Ghafari, M., “ExceptionTracer: A Solution Recommender for Exceptions in an Integrated Development Environment,” *Proceedings of the IEEE 23rd International Conference on Program Comprehension*, 2015, pp. 299-302.
11. Bacchelli, A., Ponzanelli, L., and Lanza, M., “Harnessing Stack Overflow for the IDE,” *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering*, 2012, pp. 26-30.

12. Cordeiro, J., Antunes, B., and Gomes, P., "Context-based Recommendation to Support Problem Solving in Software Development," *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering*, 2012, pp. 85-89.
13. de Souza, L., Campos, E., and de Maia, M., "Ranking Crowd Knowledge to Assist Software Development," *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 72-82.
14. Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., and Lanza, M., "Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter," *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 102-111.
15. Rahman, M., Yeasmin, S., and Roy, C., "Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions," *Proceedings of the International Conference on Software Maintenance, Reengineering, and Reverse Engineering*, 2014, pp. 194-203.
16. Chen, C., Gao, S., and Xing, Z., "Mining Analogical Libraries in Q&A Discussions — Incorporating Relational and Categorical Knowledge into Word Embedding," *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 338-348.
17. Rahman, M., Roy, C., and Lo, D., "RACK: Automatic API Recommendation Using Crowdsourced Knowledge," *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 349-359.
18. Wang, W. and Godfrey, M., "Detecting API usage obstacles: A study of iOS and Android developer questions," *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 61-64.
19. Rahman, M., Roy, C., and Keivanloo, I., "Recommending Insightful Comments for Source Code using Crowdsourced Knowledge," *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation*, 2015, pp. 81-90.
20. Chen, C., Xing, Z., and Wang, X., "Unsupervised Software-specific Morphological Forms Inference from Informal Discussions," *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 450-461.
21. Tian, Y., Lo, D., and Lawall, J., "Automated construction of a software-specific word similarity database," *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering*, 2014, pp. 44-53.
22. "The Full List of 1000 Slack Communities," URL: <https://blog.standuply.com/the-full-list-of-1000-slack-communities-2c412054ea30>. Accessed 2017.
23. Alkadhi, R., Lata, T., Guzmany, E., and Bruegge, B., "Rationale in Development Chat Messages: An Exploratory Study," *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017, pp. 436-446.

24. Chowdhury, S., and Hindle, A., "Mining StackOverflow to Filter Out Off-Topic IRC Discussion," *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 422-425.
25. Lebeuf, C., Storey, M.-A., and Zagalsky, A., "How Software Developers Mitigate Collaboration Friction with Chatbots," *Proceedings of the 20th ACM Conference on Computer-Supported Cooperative Work and Social Computing*, 2017.
26. Lin, B., Zagalsky, A., Storey, M.-A., and Serebrenik, A., "Why Developers Are Slacking Off: Understanding How Software Teams Use Slack," *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, 2016.
27. Shihab, E., Jiang, Z., and Hassan, A., "Studying the use of developer IRC meetings in open source projects," *Proceedings of the International Conference on Software Maintenance*, 2009, pp. 147-156.
28. Yu, L., Ramaswamy, S., Mishra, A., and Mishra, D., "Communications in Global Software Development: An Empirical Study Using GTK+ OSS Repository," *Proceedings of the OTM 2011 Workshops*, 2011, pp. 218-227.
29. Tang, J., Li, H., Cao, Y., and Tang, Z., "Email data cleaning," *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2005, pp. 489-498.
30. Rigby, P. and Robillard, M., "Discovering essential code elements in informal documentation," *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 832-841.
31. Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S., "Extracting structural information from bug reports," *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, 2008, pp. 27-30.
32. Cruzes, D., Mendonca, M., Basili, V., Shull, F., and Jino, M., "Automated information extraction from empirical software engineering literature: Is that possible?" *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, 2007, pp. 491-493.
33. Subramanian, S., and Holmes, R., "Making sense of online code snippets," *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 85-88.
34. Ishihara, T., Hotta, K., Higo, Y., Igaki, H., Kusumoto, S., "Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects," *Proceedings of the 19th Working Conference on Reverse Engineering*, 2012, pp. 387-391.
35. Koschke, R., "Large-scale inter-system clone detection using suffix trees," *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 309-318.

36. German, D., Di Penta, M., Gueheneuc, Y.-G., Antoniol, G., "Code siblings: Technical and legal implications of copying code between applications," *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 81-90.
37. Hemel, A. and Koschke, R., "Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices," *Proceedings of the 19th Working Conference on Reverse Engineering*, 2012, pp. 357-366.
38. Davies, J., German, D., Godfrey, M., and Hindle, A., "Software bertillonage: finding the provenance of an entity," *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 183-192.
39. Yamashina, T., Uwano, H., Fushida, K., Kamei, Y., Nagura, M., Kawaguchi, S., Iida, H., *Shinobi: A real-time code clone detection tool for software maintenance*, 26, Nara Institute of Science and Technology, 2008.
40. Keivanloo, I., Rilling, J., Charland, P., "Internet-scale real-time code clone search via multi-level indexing," *Proceedings of the 18th Working Conference on Reverse Engineering*, 2011, pp. 23-27.
41. Chen, K., Liu, P., Zhang, Y., "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 175-186.
42. Sajnani, H., Saini, V., Svajlenko, J., Roy, C., Lopes, C., "SourcererCC: Scaling code clone detection to big-code," *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157-1168.
43. Svajlenko, J., Keivanloo, I., Roy, C., "Scaling classical clone detection tools for ultra-large datasets: An exploratory study," *Proceedings of the 7th International Workshop on Software Clones*, 2013, pp. 16-22.
44. Roy, C., Zibran, M., Koschke, R., "The vision of software clone management: Past, present, and future (keynote paper)," *Proceedings of the 2014 Software Evolution Week -IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, 2014, pp. 18-33.
45. Kamiya, T., Kusumoto, S., Inoue, K., "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, 28, 7, 2002, pp. 654-670.
46. Hummel, B., Juergens, E., Heinemann, L., Conradt, M., "Index-based code clone detection: incremental, distributed, scalable," *Proceedings of the IEEE International Conference on Software Maintenance*, 2010, pp. 1-9.
47. Roy, C., Cordy, J., "Near-miss function clones in open source software: an empirical study," *Journal of Software: Evolution and Process*, 22, 3, 2010, pp. 165-189.

48. Svajlenko, J., Islam, J., Keivanloo, I., Roy, C., Mia, M., "Towards a big data curated benchmark of inter-project code clones," *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476-480.
49. Livieri, S., Higo, Y., Matushita, M., Inoue, K., "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder," *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 106-115.
50. Jiang, L., Misherghi, G., Su, Z., Glondou, S., "Deckard: Scalable and accurate tree-based detection of code clones," *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 96-105.
51. Wang, J., Li, G., Feng, J., "Can we beat the prefix filtering?: an adaptive framework for similarity join and search," *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 85-96.
52. Sarawagi, S., Kirpal, A., "Efficient set joins on similarity predicates," *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pp. 743-754.
53. Vernica, R., Carey, M., Li, C., "Efficient parallel set-similarity joins using MapReduce," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010, pp. 495-506.
54. Ambient Software Evoluton Group, "IJaDataset 2.0.," URL: <http://secold.org/projects/seclone>. Accessed June 2017.
55. Uthus, D. and Aha, D., "Multiparticipant Chat Analysis: A Survey," *Artificial Intelligence*, 199, 2013, pp. 106-121.
56. Elsner, M. and Charniak, E., "You talking to me? A Corpus and Algorithm for Conversation Disentanglement," *Proceedings of the Association of Computational Linguistics: Human Language Technology*, 2008, pp. 834-842.
57. Stack Exchange, "Stack Exchange Data Dump," URL: <https://archive.org/details/stackexchange>. Accessed August 27, 2017.
58. Richardson, L., "Beautiful Soup," URL: <https://www.crummy.com/software/BeautifulSoup>. Accessed May 7, 2017.
59. Bird, S., Loper, E., and Klein, E., *Natural Language Processing with Python*, O'Reilly Media Inc, 2009.
60. Elsner, M. and Charniak, E., "Disentangling chat," *Computational Linguistics*, 36, 3, 2010, pp. 389-409.
61. Fox, C., "Lexical Analysis and Stoplists," *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, 1992.
62. "pdftotext online tool," URL: <http://pdftotext.com>. Accessed January 30, 2017.

63. "convertmypdf online tool," URL: <http://www.convertmypdf.net/>. Accessed January 30, 2017.
64. "convertpdf totext online tool," URL: <http://www.convertpdf totext.net/>. Accessed January 30, 2017.
65. "ocrconvert ocr tool," URL: <http://www.ocrconvert.com/>. Accessed January 30, 2017.
66. "abbyyfinereader ocr tool," URL: <https://www.abbyy.com/en-us/finereader/>. Accessed January 30, 2017.
67. "Apache Lucene," URL: <https://lucene.apache.org/core/>. Accessed June 2017.
68. Svajlenko, J., Keivanloo, I., Roy, C., "Big data clone detection using classical detectors: an exploratory study," *Journal of Software: Evolution and Process*, 27, 6, 2015, pp. 430-464.
69. Svajlenko, J., Roy, C., "Evaluating clone detection tools with BigCloneBench," *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 131-140.
70. Svajlenko, J., Roy, C., "Evaluating modern clone detection tools," *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 321-330.
71. Gode, N., Koschke, R., "Incremental clone detection," *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 219-228.
72. Roy, C., Cordy, J., "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008, pp. 172-181.
73. Kim, S., Woo, S., Lee, H., Oh, H., "Vuddy: A scalable approach for vulnerable code clone discovery," *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.
74. Sheneamer, A., Kalita, J., "A survey of software clone detection techniques," *International Journal of Computer Applications*, 137, 10, March 2016.
75. Svajlenko, J., "BigCloneEval," URL: <https://github.com/jeffsvajlenko/BigCloneEval>. Accessed June 2017.
76. "BigCloneBench," URL: <https://github.com/clonebench/BigCloneBench>. Accessed June 2017.
77. Hindle, A., Barr, E., Su, Z., Gabel, M., Devanbu, P., "On the naturalness of software," *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 837-847.

APPENDIX

The publications and presentations for this project include:

- Shepherd, D. (Presenter), “Creating a Web-Scale Repository of Documented & Analyzable Code Snippets from Developer Communications,” *DARPA MUSE PI Meeting*, Santa Cruz, California, USA, September 7, 2016.
- Chatterjee, P., Nishi, M.A., Damevski, K., Augustine, V., Pollock, L., and Kraft, N., “What Information about Code Snippets Is Available in Different Software-Related Documents? An Exploratory Study,” *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, Klagenfurt, Austria, 2017.
- Chatterjee, P., Gause, B., Hedinger, H., and Pollock, L., “Extracting Code Segments and Their Descriptions from Research Articles,” *Proceedings of the 14th International Conference on Mining Software Repositories*, Buenos Aires, Argentina, 2017.
- Kraft, N. (Presenter), “Creating a Web-Scale Repository of Documented & Analyzable Code Snippets from Developer Communications,” *DARPA MUSE PI Meeting*, Washington, D.C., USA, May 10, 2017.
- Nishi, M.A., and Damevski, K., “Scalable Code Clone Detection and Search based on Adaptive Prefix Filtering,” *Journal of Systems and Software*, 137, 2017, pp.130-142.
- Kraft, N. (Presenter), “Understanding the Quality of Code Snippets Mined from Q&A Chat Sessions,” *DARPA MUSE PI Meeting*, Austin, Texas, USA, January 22, 2018.
- Greco, C., Haden, T., Damevski, K., “StackInTheFlow: Behavior-Driven Recommendation System for Stack Overflow Posts,” *Proceedings of the International Conference on Software Engineering*, Gothenburg, Sweden, 2018.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACM	Association for Computing Machinery
API	application programming interface
DARPA	Defense Advanced Research Projects Agency
DL	digital library
IEEE	Institute of Electrical and Electronics Engineers
HTML	hypertext markup language
IDE	integrated development environment
IRC	internet relay chat
LDA	latent Dirichlet allocation
LOC	lines of code
MT3	Moderately Type-3
MUSE	Mining and Understanding Software Enclaves
NLP	natural language processing
NLTK	natural language toolkit
OCR	optical character recognition
Q&A	question and answer
SE	software engineering
ST3	Strongly Type-3
URL	uniform resource locator
VST3	Very Strongly Type-3
WT3/T4	Weakly Type-3
XML	extensible markup language