2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

Extracting Code Segments and Their Descriptions from Research Articles

Preetha Chatterjee, Benjamin Gause, Hunter Hedinger, and Lori Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716 USA
Email: {preethac, bengause, hedinger, pollock}@udel.edu

***Abstract*—The availability of large corpora of online software-related documents today presents an opportunity to use machine learning to improve integrated development environments by first automatically collecting code examples along with associated descriptions. Digital libraries of computer science research and education conference and journal articles can be a rich source for code examples that are used to motivate or explain particular concepts or issues. Because they are used as examples in an article, these code examples are accompanied by descriptions of their functionality, properties, or other associated information expressed in natural language text. Identifying code segments in these documents is relatively straightforward, thus this paper tackles the problem of extracting the natural language text that is associated with each code segment in an article. We present and evaluate a set of heuristics that address the challenges of the text often not being colocated with the code segment as in developer communications such as online forums.**

***Index Terms*—mining software repositories, information extraction, code snippet description, text analysis**

I. Introduction

With the increased online sharing of software-related information, software engineers often look beyond documentation and their local resources, seeking examples and advice from experiences of other developers not geographically nearby. The examples are more useful if there is an explanation of their functionalities and properties that they exhibit. These code descriptions are often not found in others' source code, but are instead in other software-related artifacts such as Q&A forums, blog posts, and emails. The vast availability of online resources has also motivated researchers to develop techniques to help developers more efficiently locate code examples with descriptions by automatically mining code examples from various sources, including emails [1], [2], [3], [4], Q&A forums [5], [6], [7], [8], API documentation [9], bug reports [10], and stack traces.

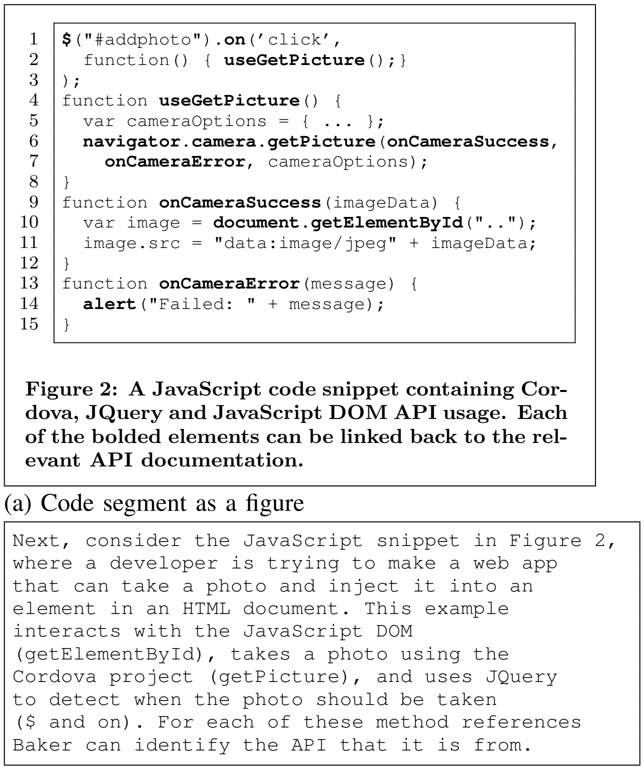
Digital libraries for computer science research and education articles could potentially provide a large amount of code examples with descriptions. The ACM Digital Library contains an archive of every article and publication published by ACM from 1950s to present [11]. The IEEE Xplore DL includes over 180 journals, over 1,400 conference proceedings, more than 3,800 technical standards, over 1,800 eBooks and over 400 educational courses. Each month, 20,000 new documents are added to IEEE Xplore on average [12]. The publication count of the top conference in the field of software engineering alone, ICSE, is 8,459 at present [13]. In total, the IEEE Xplore digital library provides web access to more than 3.5-million full-text documents of publications in the fields of electrical engineering, computer science and electronics [12].

This paper explores the potential for digital libraries of computer science research and education conference and journal articles to serve as another resource for good code examples with descriptions. To investigate the availability of code examples in computer science digital libraries, we manually counted the number of code segments in 100 randomly selected research articles from ICSE, FSE, and ICSME proceedings. 70% of the selected articles contained one or more code segments, with an average of 3-4 code segments per article. The examples always have some associated descriptions of their functionality, properties, or other associated information expressed in natural language text.

As an example of the kind of information that can be extracted from source code descriptions in research literature, consider a code snippet and its description in Figure 1, extracted from a paper published in ICSE 2014. The description of the code snippet provides useful information about the source code, including (1) the programming language it was written in, (2) the intent of the overall code that the programmer is implementing of which this code segment is a part of (i.e., a web application), (3) some of the APIs it uses, and (4) the sub-steps being implemented by the code segment, i.e., a description of its functionality.

Mining code segments 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 segments 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 segments are embedded in the mainstream text. In contrast, code segments 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 segment from description makes the description identification problem, i.e., the problem of identifying all the

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.10



```
$("#addphoto").on(’click’,
  function() { useGetPicture();}
);
function useGetPicture() {
  var cameraOptions = { ... };
  navigator.camera.getPicture(onCameraSuccess,
    onCameraError, cameraOptions);
}
function onCameraSuccess(imageData) {
  var image = document.getElementById("..");
  image.src = "data:image/jpeg" + imageData;
}
function onCameraError(message) {
  alert("Failed: " + message);
}
```

Figure 2: A JavaScript code snippet containing Cordova, JQuery and JavaScript DOM API usage. Each of the bolded elements can be linked back to the relevant API documentation.

(a) Code segment as a figure

```
Next, consider the JavaScript snippet in Figure 2,
where a developer is trying to make a web app
that can take a photo and inject it into an
element in an HTML document. This example
interacts with the JavaScript DOM
(getElementById), takes a photo using the
Cordova project (getPicture), and uses JQuery
to detect when the photo should be taken
($ and on). For each of these method references
Baker can identify the API that it is from.
```

(b) Code-related descriptive text

Fig. 1: A sample code snippet with description extracted from the article “Live API Documentation” (ICSE 2014)

text that contains description of the functionality or property of an embedded code segment, more difficult. The problem is further complicated by the common situation in which the research article contains multiple code segments, in which case the textual descriptions that are identified need to be mapped to the corresponding code segment. 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 segment and description mining technique, the technique cannot analyze every line of text.

Researchers have developed techniques to automatically extract code segments from emails, bug reports, Q&A sites and tutorials [1], [2], [10], [5]. These techniques are also applicable to extract code segments 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 API is being described [5], [14]. Text preceding code segments in Stack Overflow can be extracted as potential comments for similar code segments in an application [6]. Similarly, method descriptions can be extracted using clues in the text [15], [16].

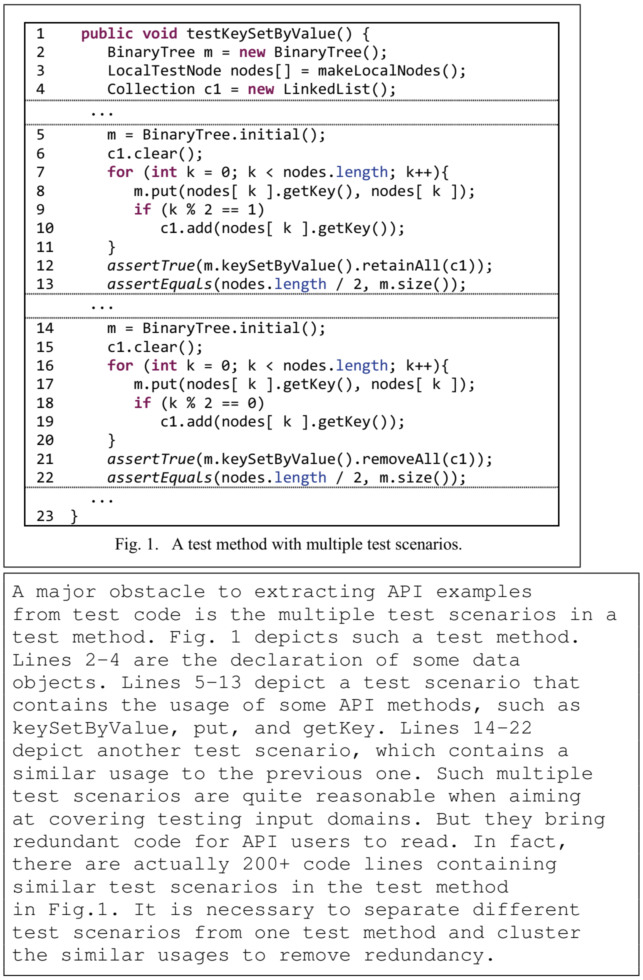
To the best of our knowledge, this is the first paper to address the code description identification problem for research articles and other similar documents such as dissertations. The main contributions of this paper are:

- a set of heuristics to automatically identify and map text that is describing code segments in research articles, including segments embedded as figures
- a set of 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 XML-based representation with markups to associate identified code segments with their corresponding descriptions
- an evaluation study that evaluates the effectiveness of the presented code description identification techniques

II. Motivating Examples

In addition to the example in Figure 1, we present three additional code snippets and their descriptive text extracted from research articles and discuss how they could be used to further motivate extracting code segments with descriptions from research articles. The description of the code snippet shown in Figure 2 explains this code's inefficiency and provides useful information including (1) it is a method used for testing more than one test scenario, (2) the specific test coverage of the method, (3) it contains redundancy in the source code, (4) it contains example usage of API methods, and (5) a proposed solution to remove the redundancy. Beyond using the code and its description to show usage of particular APIs in text methods, using this example code segment to learn about redundancies in test methods, an IDE could be designed to detect redundancy in lines of code, prompting the user to make separate methods for different test scenarios and congregate the ones with similar functionalities.

The next description in Figure 3 indicates that the code segment demonstrates a common memory leak pattern the C programming language. The description of the code snippet provides useful information, including (1) the programming language of the source code, (2) a comment description of the pattern of recurring memory leaks, (3) the functionality of the procedure shown, (4) functionalities of individual method calls within the procedure, (5) the type of database used by the program, and (6) the reason of the memory leak. By mining such code segments and their descriptions, a C programming language tutorial could include this code in a lesson on fixing recurring memory leaks, or an IDE could be extended to identify such memory leak patterns as they are implemented. From this example, the consequences of using unsuitable loop exit statements also can be evaluated and avoided. The last example code description shown in Figure 4 indicates that the code snippet depicts a security vulnerability in a typical SQL injection. The description of the code snippet specifies (1) the functionalities of specific SQL queries, (2) the security issues inherent of the vulnerable code, (3) the vulnerability in the code. Without looking carefully, a reader would not be able to understand the security vulnerability in this example code without the corresponding description. This code and its



```
1    public void testKeySetByValue() {
2       BinaryTree m = new BinaryTree();
3       LocalTestNode nodes[] = makeLocalNodes();
4       Collection c1 = new LinkedList();
     ...
5       m = BinaryTree.initial();
6       c1.clear();
7       for (int k = 0; k < nodes.length; k++){
8          m.put(nodes[ k ].getKey(), nodes[ k ]);
9          if (k % 2 == 1)
10            c1.add(nodes[ k ].getKey());
11      }
12      assertTrue(m.keySetByValue().retainAll(c1));
13      assertEquals(nodes.length / 2, m.size());
     ...
14      m = BinaryTree.initial();
15      c1.clear();
16      for (int k = 0; k < nodes.length; k++){
17         m.put(nodes[ k ].getKey(), nodes[ k ]);
18         if (k % 2 == 0)
19            c1.add(nodes[ k ].getKey());
20      }
21      assertTrue(m.keySetByValue().removeAll(c1));
22      assertEquals(nodes.length / 2, m.size());
     ...
23 }
```

Fig. 1. A test method with multiple test scenarios.

A major obstacle to extracting API examples from test code is the multiple test scenarios in a test method. Fig. 1 depicts such a test method. Lines 2-4 are the declaration of some data objects. Lines 5-13 depict a test scenario that contains the usage of some API methods, such as keySetByValue, put, and getKey. Lines 14-22 depict another test scenario, which contains a similar usage to the previous one. Such multiple test scenarios are quite reasonable when aiming at covering testing input domains. But they bring redundant code for API users to read. In fact, there are actually 200+ code lines containing similar test scenarios in the test method in Fig.1. It is necessary to separate different test scenarios from one test method and cluster the similar usages to remove redundancy.

Fig. 2: A sample code snippet with description Excerpt from the paper "Mining API Usage Examples from Test Code" (ICSME '14)

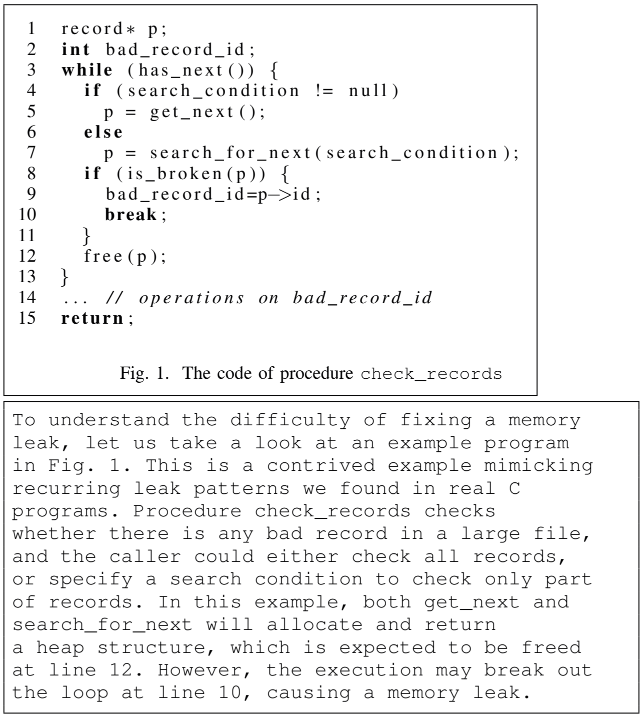
description could serve as a good lesson in a SQL tutorial or be used as a pattern in a vulnerability detection tool that provides feedback about the vulnerabilities found, based on this description text.

These examples are just a sampling of the kind of information that can be learned about mined code snippets in research articles if the descriptive text can be mined along with the code snippets to provide the writer's perspective on the properties and functionalities of the mined code segments. Because they originate in research papers, the descriptions typically include the functionality and properties, and are used commonly as examples of those properties, which is a rich kind of information for many purposes.

III. Approach

A. Overview of CoDesNPub Miner

Figure 5 presents the phases of our whole process for automatic preprocessing, classification, identification and markup of research articles. 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 [17], convertmypdf [18], convertpdftotext [19], etc. making sure they can handle both single and double column formatting of articles. Since some code segments are embedded into articles as images, this phase leverages existing image-to-text converters such as ocrconvert [20], abbyyfinereader [21], etc. to process the images into text so all code segments can be identified.



```
1  record* p;
2  int bad_record_id;
3  while (has_next()) {
4    if (search_condition != null)
5      p = get_next();
6    else
7      p = search_for_next(search_condition);
8    if (is_broken(p)) {
9      bad_record_id=p->id;
10     break;
11   }
12   free(p);
13 }
14 ... // operations on bad_record_id
15 return;
```

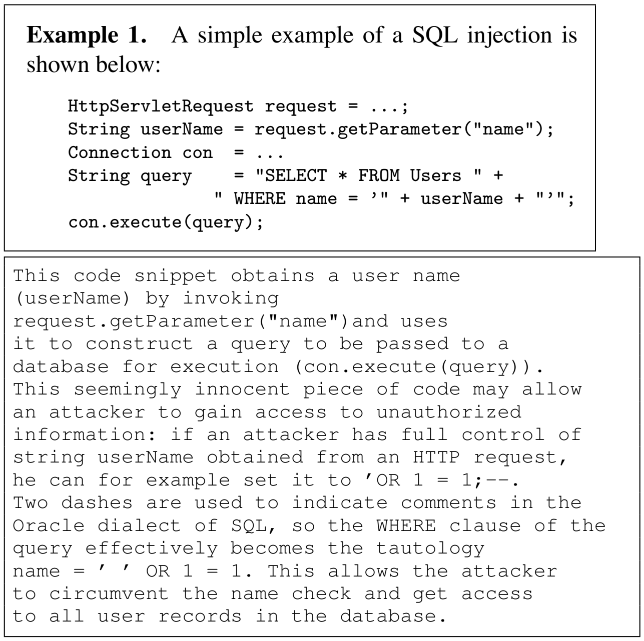
Fig. 1. The code of procedure check_records

To understand the difficulty of fixing a memory leak, let us take a look at an example program in Fig. 1. This is a contrived example mimicking recurring leak patterns we found in real C programs. Procedure check_records checks whether there is any bad record in a large file, and the caller could either check all records, or specify a search condition to check only part of records. In this example, both get_next and search_for_next will allocate and return a heap structure, which is expected to be freed at line 12. However, the execution may break out the loop at line 10, causing a memory leak.

Fig. 3: A sample code snippet with description Excerpt from the paper "Safe Memory-leak Fixing for C Programs" (ICSE '15)

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 [10], [1], [4], [5]. In this paper, we use 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 segments.

The main focus of this paper is 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 segment 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



Example 1. A simple example of a SQL injection is shown below:

```
HttpServletRequest request = ...;
String userName = request.getParameter("name");
Connection con  = ...
String query    = "SELECT * FROM Users " +
                " WHERE name = ’" + userName + "’";
con.execute(query);
```

```
This code snippet obtains a user name
(userName) by invoking
request.getParameter("name")and uses
it to construct a query to be passed to a
database for execution (con.execute(query)).
This seemingly innocent piece of code may allow
an attacker to gain access to unauthorized
information: if an attacker has full control of
string userName obtained from an HTTP request,
he can for example set it to ’OR 1 = 1;--.
Two dashes are used to indicate comments in the
Oracle dialect of SQL, so the WHERE clause of the
query effectively becomes the tautology
name = ’ ’ OR 1 = 1. This allows the attacker
to circumvent the name check and get access
to all user records in the database.
```

Fig. 4: A sample code snippet with description Excerpt from the paper "Finding Security Vulnerabilities in Java Applications with Static Analysis" (SSYM'05)

structural features of the text and embedded code segments 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 segments and related seeds and neighbors marked up, or a set of extracted code segments and related seeds and neighbors for a database of mined code examples with descriptive information.

B. Code Description Identification

To develop our automatic code description identification technique, we analyzed the text of randomly selected computer science research articles from ACM and IEEE digital libraries, which collectively included over 200 code examples. Based on our manual inspection of both text related to the code segments and text not related to the code segments, 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.

References Figure Containing Code. While some code segments 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 very accurate cues for an automatic system to identify sentences related to the code segment, when we are able to identify that the figure being referenced is indeed a code segment.

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. For example in Figure 1, *ReferencesCodeFigure* would identify the sentence, *"Next, consider the JavaScript snippet in Figure 2, where a developer is trying to make a web app that can take a photo and inject it into an element in an HTML document."*

Located Immediately Before or After Inlined Code. Sometimes, authors of research articles place their code examples directly inlined within the running text, similar to the use of code segments in online forums and emails. When this occurs, it is most likely that they are discussing the code segment in sentences just before or just after (or both before and after) the code segment itself.

The *TextBefore* and *TextAfter* heuristics identify the sentences immediately before and immediately after any inlined code segment as potential code descriptions, respectively. For example, in Figure 2, *TextAfter* would extract *"A major obstacle to extracting API examples from test code is the multiple test scenarios in a test method."*, since this is the sentence that occurs immediately after the code segment in the document.

Without combining with other heuristics, these heuristics can be inaccurate if the author always describes code segments before or after and not both locations. These heuristics capture the relative location only of sentences surrounding inlined code. Additional sentences in the nearby location will be considered by the neighborhood sentence identification heuristics.

Contains Code Identifiers. Sentences describing code segments in a research article often contain code identifiers from the associated code segment. 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 segments in a research article that contain a word that also appears in any of the code segments as a user-defined identifier. This heuristic requires tokenization of the code segments within the document, creation of a dictionary of variable, method, and class names for each code segment, and removal of keywords from those dictionaries. The heuristic identifies and maps sentences to associated code segments based on occurrences of the dictionary names in the sentence.

For example in Figure 3, *ContainsCodeIdentifiers* would identify the sentence *"In this example, both **get_next** and **search_for_next** will allocate and return a heap structure, which is expected to be freed at line 12."*, based on the presence of the code identifiers "get_next" and "search_for_next" in the code segment described by this sentence. This heuristic has the potential to be inaccurate when code segments use identifiers that are commonly used as regular words in sentences.

References Code By Position. Authors somtimes use specific cue words or phrases pertaining to software engineering and development when describing code segments in research articles. *ReferencesCodeByPosition* identifies the sentences that

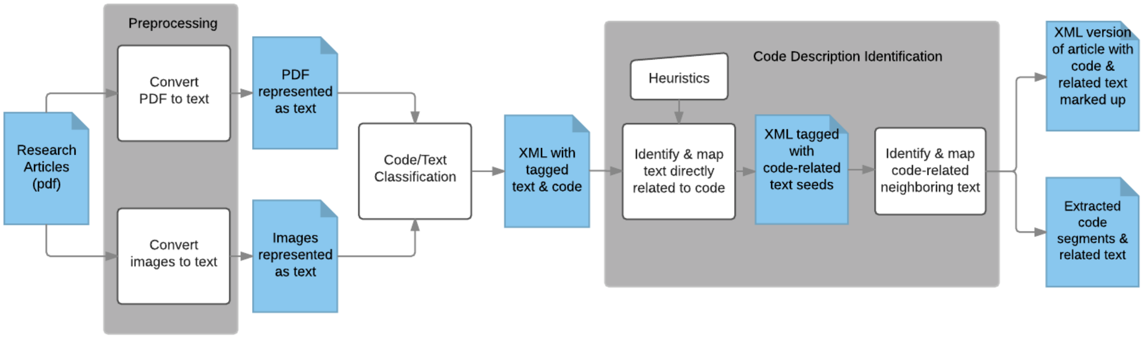

Preprocessing
Research Articles (pdf)
Convert PDF to text
Convert images to text
PDF represented as text
Images represented as text
Code/Text Classification
XML with tagged text & code
Code Description Identification
Heuristics
Identify & map text directly related to code
XML tagged with code-related text seeds
Identify & map code-related neighboring text
XML version of article with code & related text marked up
Extracted code segments & related text

Fig. 5: Overview of CoDesNPub Miner

have specific cue words or phrases that suggest that a sentence is describing a code segment in the document. This heuristic looks for phrases such as "...in the following code...", "in the running example", etc. Specifically, this heuristic aims to identify sentences containing code-indicating words such as 'code', 'method', 'loop', 'Javascript', etc. Based on the adjective in the phrase, it searches either before or after the sentence for a code segment located in the designated relative position near the neighboring two paragraphs to the sentence, to confirm that the sentence is referring to a code segment. For example, in Figure 4, *ReferencesCodeByPosition* would identify the sentence, *"**This code snippet** obtains a user name (userName) by invoking request.getParameter("name")and uses it to construct a query to be passed to a database for execution (con.execute (query))."*

1) Putting It All Together: 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 segment, 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 segment.

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 perform 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 evaluate the various precisions and numbers of sentences identified with different thresholds and scoring schemes.

2) Identifying Neighboring Code-related Text: 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, etc. and are important for better understanding and reuse of the same code example. Figure 4 shows an example where the code-related neighboring text would be the sentence *"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 that example code, but the heuristics with the textual cues would not identify this sentence.

Our manual analysis suggested that that if a part of a paragraph of text contains sentences identified by the heuristics to describe a code segment, then the entire paragraph often describes the code segment 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.

C. *Code Description Identification Example*

Consider a paragraph extracted from a paper published in ICSE 2014. 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.5 shows a typical test method of this pattern.
The method tests a set of basic functionality of
API class BasicAuthCache, including the method
put, get, remove and clear. There are three test
scenarios in the method: line  4-5,  line 6-7,
line 8-10. They share two data objects, cache and
authScheme. Their method invocation sequences
are not same and there is no unified test target
method. But there is a common subsequence
among three method invocation sequences, i.e.,
the invocations of get and HttpHost.
```

Excerpt from the paper "Mining API Usage Examples from Test Code" (ICSME '14)

ReferencesCodeFigure would identify the first sentence as code description due to the presence of the word *"Fig.5"*, which is the figure number for the code segment described in this paragraph. We would assign a score of 3 to this sentence. The next sentence contains code identifiers *"BasicAuthCache"* and *"get"*, found in the described code segment 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 or our heuristics. The fourth sentence would be identified by *ContainsCodeIdentifiers* since it contains the code identifiers *"cache"* and *"authScheme"*, found in the described code segment. Hence, this sentence would be assigned a score of 2. 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 *"HttpHost"*. However, the same sentence would also get an additional score by *TextBefore* since this sentence is found immediately above the code segment 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.

IV. Evaluation

We designed our evaluation to answer the research question:

RQ1: How effective is our approach to automatically identify code descriptions in natural language text of research articles?

In addition, we also collected data to answer two questions about how code segments are described in research articles. Namely, we collect data to answer:

RQ2: What kinds of information are available in natural language text describing code segments in research articles?

RQ3: How do authors typically reference code segments within their code description text in research articles (i.e., What cues are most prominent?)

A. *Evaluation Design*

1) Implementation: 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 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.

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

To answer RQ1, we measure the effectiveness of the overall precision and recall of the code description identification, and also the precision of the seed identification. We do 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 of the overall code description identification is computed by determining the percentage of all the sentences that describe the code segments in the study (as identified by human judges) that are also identified as code descriptions by the automatic technique.

To answer RQ2, 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 RQ3, one of the authors used the results from a previous study [22] 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 twenty sub-labels for the observed code properties as described in Table IV. RQ3 is addressed by coding each annotated sentence and computing the frequency of occurrence of each label in the subject set of research articles.

3) Methodology: 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 on this paper, 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 segment. Based on the timing results, each of the ten judges was assigned research papers for 20-30 of the randomly selected code segments. To account for potential subjectivity of human opinion, each of the 100 code segments 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 segment or any property of the code segment (highlighted in yellow in the document), and label each highlighted text with the related code segment 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 segments 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.

B. Results and Discussion

We organize our evaluation results by research questions.

RQ1: How effective is our approach to automatically identify code descriptions in natural language text of research articles?

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 are 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 I presents results to answer the first two research subquestions, 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.

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 I: Precision of seed heuristics (Scoring: References Figure Containing Code:3, Located Immediately Before or After Inlined Code:1, Contains Code Identifiers, References Code By Position:2)

Minimum # of Seeds	Precision	Recall
1-24%	39.05	70.20
≥ 25%	53.41	50.33
≥ 50%	66.04	28.45
≥ 75%	68.30	20.53

TABLE II: Effectiveness of code description identification with different schemes to identify neighboring code-related text

Table II 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 II 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 (≥50% and ≥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.

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 III 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 III, 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 segment specifically, it gives us some information about the implementation, which might be useful

Identified correctly as code descriptions
First, we notice that EVOSUITE uses the method toString rather than getRootElementName 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., (σ1, 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.

TABLE III: Examples of Analyzed Code Description Sentences

for building code recommendation systems.

Our analysis of the sentences where CoDesNPub Miner-missed sentences describing code segments revealed some limitations of using a system based only on features of phrases contained in sentences. The fifth example in Table III 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.

RQ2: What kinds of information are available in natural language text describing code segments in research articles?

Figure 6a and Figure 6b depict our 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 6b 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 segment is implemented in a particular way, which could be valuable meta-data for a mined code example for learning.

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 segments in digital libraries.

Labels	Sub-Labels	Description
Design	Programming Language	Programming language
	Framework	Framework used
	Time/Space Complexity	Code complexity
Structure	Data Structure	Data structures or variable types
	Control Flow	Types of control statements used
	Data Flow	Data flow chains included
	Lines of code	Length of code
Explanatory	Rationale	Why being implemented in this way
	Functionality	What is being implemented
	Methodology	How functionality is implemented
	Output of code	Results of running code
	Similarity	Syntactic or semantically similar code blocks
	Modification	Change(s) to existing code
Clarity	High	Code is clean and understandable
	Low	Code is unclear or overly complex
Efficiency	Efficient	Better/efficient code example
	Inefficient	Inefficient code example
	Assumptions	Conditions to be met to ensure correctness
Erroneous	Compilation	Code that fails to compile
	Runtime	Contains runtime errors or exceptions thrown

TABLE IV: Description of labels and sublabels

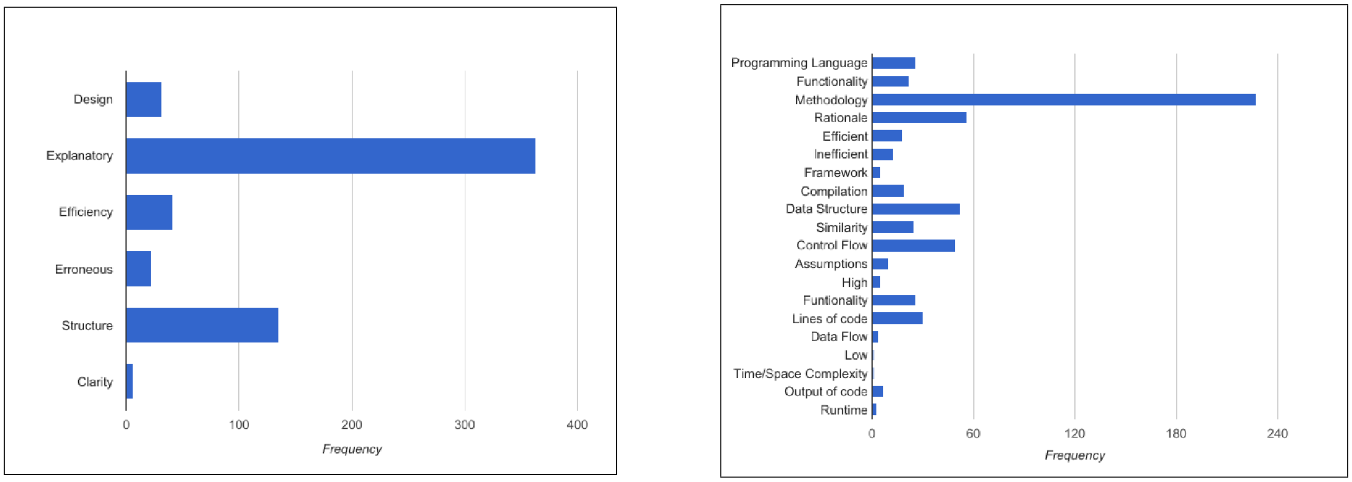
RQ3: How do authors typically reference code segments within their code description text in research articles (i.e., What cues are most prominent?)

The relative frequency of each feature used to indicate code description text is shown in Figure 7, which depicts that *References Figure Containing Code* is the most prevalent heuristic. The next prevalent heuristic is *Neighboring Code-related Text* which helps in identifying sentences that describe less obvious details about code segments.

C. Threats to Validity

Our subjects are selected from both journal and conference papers in software engineering, across different years from ACM DL and 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 segments.

One possible threat could be programming language dependence. The technique we used to identify code segments



(a) Frequency of labels

(b) Frequency of sub-labels

Fig. 6: Kinds of information in research articles

in unstructured documents is capable of identifying code segments in different programming languages from documents containing code segments 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 segments. 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, Python, etc.

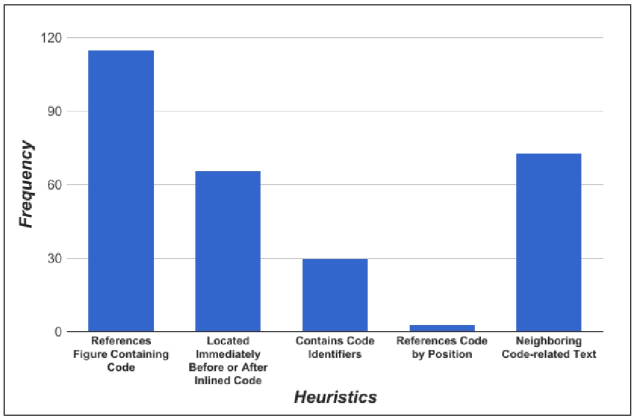


Fig. 7: Relative frequency of each feature indicating code description text

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 segments that are used only as examples in an empirical study. Our datasets for development and evaluation consist of papers that do not contain pseudocode. We plan to extend our approach to identify both code fragments and pseudocode in our future work.

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 segments. 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 segment 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 segments in our evaluation set might lead to different results. However, we needed to make the human annotation work reasonable to recruit judges. We will expand the evaluation study in the near future with more participants, and research papers containing more code segments.

V. Related Work

The most related work to this research is in collecting and analyzing information from sets of research articles, identification of code snippets from unstructured documents, and identification of any textual descriptions associated with embedded code snippets.

Analyzing Collections of Research Articles. Cruzes et. al [14] ran an entity recognition tool called Site Content Analyzer on software engineering papers to analyze the linguistic features of the documents such as word density and frequency. Based on the results, they claim that information extraction techniques like text mining can support systematic reviews and creation of repositories of SE empirical evidence. Researchers have analyzed repositories of research articles to support their evaluations. For example, Siegmund et. al [23] discussed the tradeoff between internal and external validity and replication, complemented with a literature review about the status of empirical research in software engineering. Kampenes et. al [24], [25] reported systematic reviews of controlled and quasi-experiments published in major software engineering proceedings. They investigated the selection bias, practice of effect size reporting, summarized standardized effect sizes detected in the experiments, and provided advice for improvements based on the results. Tichy et. al [26] discussed the lack of experimentally validated results and quantitative evaluations in computer science journals, supported by a survey of 400 research articles.

Code Segment Extraction. Bacchelli et. al. [1] used lightweight regular expression-based techniques to identify code blocks in emails. Their features were lexical, focusing on programming language specific characteristics such as special characters and keywords and end of line markers. Their evaluation suggests that using lightweight methods are to be preferred over heavyweight techniques for source code extraction from emails. Tang et al. [2] filtered out non-NL text including email headers, signatures, and code-related content (stack traces, patches, and source code snippets) before cleaning the remaining text with paragraph and sentence detection. They manually labeled data sets and then used SVM classification with specific features for each filtering target. Cerulo et. al [3] introduced an approach, based on Hidden Markov Models (HMMs), to extract coded information islands, such as source code, stack traces, and patches, from emails. They trained a HMM for each category of information contained in the text of the emails, and used the Viterbi algorithm to recognize whether the sequence of tokens observed in a text switches among those HMMs. Evaluation showed an accuracy of 82%-99%. This approach does not require manual definition of regular expressions or parsers.

Bettenburg et al. [10] developed a tool called InfoZilla that identifies and classifies code patches, stack traces, source code, and enumerated lists in bug reports. They apply specific filters for each category, using island parsing for identifying source code. Evaluation showed almost perfect accuracy for each kind of structure. The approach focuses on bug reports, all with the same programming language used in the code snippets, patches, and stack traces, and would require developing new parsers to handle a broader class of developer documentation.

Subramanian et. al. [16] performed analyses of source code snippets found in Stack Overflow, constructing an Abstract Syntax Tree (AST) for each code snippet and then parsing to effectively identify specific API usage. Building on their previous work [16], Subramanian et. al. [6] developed an iterative, deductive method of linking source code examples to API documentation. Rigby et. al [5] developed a tool that uses an island parser to identify code elements in a Stack Overflow post. Evaluation on documents that contain over 7058 distinct tags on StackOverflow showed an average precision and recall of 0.92 and 0.90, respectively. These techniques are also applicable to extract code segments from research articles.

Code Description Identification. Panichella et. al [4] developed a feature-based approach to automatically extract method descriptions from developer communications in bug tracking systems and mailing lists. Evaluation on two open source systems indicated that the approach is able to extract method descriptions with a precision up to 79% for Eclipse and 87% for Lucene. Vassallo et. al [27] built on their previous work [4], to design a tool that extracts candidate method documentation from StackOverflow discussions, and creates Javadoc descriptions. Their tool is able to extract descriptions for 20% and 28% of the Lucene and Hibernate methods with a precision of 84% and 91% respectively.

Wong et. al [7] proposed an automatic comment generation approach, which mines comments from Stack Overflow, and uses the code-description mappings in the posts to automatically generate descriptive comments for similar code segments matched in open-source projects. For Java and Android tagged Q & A posts, they extracted 132,767 code-description mappings, to generate 102 comments automatically for 23 Java and Android projects. Rahman et. al [28] developed a heuristic-based technique for mining comments from Stack Overflow Q & A site for a given code segment. Evaluation on 292 Stack Overflow code segments and 5,039 discussion comments showed that their approach has a recall of 85.42%. Most of these systems focused on identifying source code descriptions from Stack Overflow posts, where the text describing the code is always found next to the code snippet. StackOverflow posts also have specific XML-tagged formats, which makes the extraction of the information straightforward.

VI. Conclusion And Future Work

This paper takes a first step towards unleashing the potential to mine the vast number of computer science articles in digital libraries for code segments that come with useful descriptive information about their functionality and properties. We present and evaluate the first technique to automatically identify natural language descriptions of code segments embedded within articles, where code segments can be separated as figures that are not located next to their descriptive text. Our evaluation study indicates 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 segments could be learned.

Future work includes fully automating the front-end preprocessing of articles, more extensive evaluation and study with other types of articles and different domains, and more research to improve the precision and recall of the automated description identification.

Acknowledgment

This research is supported by the National Science Foundation under Grant No.1422184 and the DARPA MUSE program under Air Force Research Lab contract no. FA8750-16-2-0288.

References

[1] A. Bacchelli, M. D'Ambros, and M. Lanza, "Extracting source code from e-mails," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, June 2010, pp. 24–33.

[2] J. Tang, H. Li, Y. Cao, and Z. Tang, "Email data cleaning," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: ACM, 2005, pp. 489–498. [Online]. Available: http://doi.acm.org/10.1145/1081870.1081926

[3] L. Cerulo, M. Ceccarelli, M. Di Penta, and G. Canfora, "A hidden markov model to detect coded information islands in free text," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, Sept 2013, pp. 157–166.

[4] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, June 2012, pp. 63–72.

[5] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 832–841. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486897

[6] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 643–652. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568313

[7] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 562–567.

[8] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 392–403. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884800

[9] J. Montandon, H. Borges, D. Felix, and M. Valente, "Documenting apis with examples: Lessons learned with the apiminer platform," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 401–408.

[10] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 27–30. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370757

[11] "ACM wiki page," https://en.wikipedia.org/wiki/ Association_for_Computing_Machinery.

[12] "IEEEXplore wiki page," https://en.wikipedia.org/wiki/IEEE_Xplore.

[13] "ICSE publication history," http://dl.acm.org/event.cfm?id=RE228 &tab=pubs&CFID=723067040&CFTOKEN=52119863.

[14] D. Cruzes, M. Mendonça, V. Basili, F. Shull, and M. Jino, "Automated information extraction from empirical software engineering literature: Is that possible?" in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 491–493. [Online]. Available: http://dl.acm.org/citation.cfm?id=1302496.1302980

[15] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining api types using text classification," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 869–879. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818859

[16] S. Subramanian and R. Holmes, "Making sense of online code snippets," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, May 2013, pp. 85–88.

[17] "pdftotext online tool," http://pdftotext.com.

[18] "convertmypdf online tool," http://www.convertmypdf.net/.

[19] "convertpdftotext online tool," http://www.convertpdftotext.net/.

[20] "ocrconvert ocr tool," http://www.ocrconvert.com/.

[21] "abbyyfinereader ocr tool," https://www.abbyy.com/en-us/finereader/.

[22] P. Chatterjee, M. Nishi, K. Damevski, V. Augustine, L. Pollock, and N. Kraft, "What information about code snippets is available in different software-related documents? an exploratory study," in *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'17)*, Feb. 2017.

[23] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 9–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818759

[24] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "Systematic review: A systematic review of effect size in software engineering experiments," *Inf. Softw. Technol.*, vol. 49, no. 11-12, pp. 1073–1086, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2007.02.015

[25] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of quasi-experiments in software engineering," *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 71–82, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.04.006

[26] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz, "Experimental evaluation in computer science: A quantitative study," *J. Syst. Softw.*, vol. 28, no. 1, pp. 9–18, Jan. 1995. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(94)00111-Y

[27] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora, "Codes: Mining source code descriptions from developers discussions," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 106–109. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597799

[28] M. Rahman, C. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, Sept 2015, pp. 81–90.