Recommending Insightful Comments for Source Code using Crowdsourced Knowledge

*Mohammad Masudur Rahman *Chanchal K. Roy †Iman Keivanloo
*University of Saskatchewan, Canada †Queen's University, Canada
{*masud.rahman, *chanchal.roy}@usask.ca, †iman.keivanloo@queensu.ca

***Abstract*—Recently, automatic code comment generation is proposed to facilitate program comprehension. Existing code comment generation techniques focus on describing the functionality of the source code. However, there are other aspects such as insights about quality or issues of the code, which are overlooked by earlier approaches. In this paper, we describe a mining approach that recommends insightful comments about the quality, deficiencies or scopes for further improvement of the source code. First, we conduct an exploratory study that motivates crowdsourced knowledge from Stack Overflow discussions as a potential resource for source code comment recommendation. Second, based on the findings from the exploratory study, we propose a heuristic-based technique for mining insightful comments from Stack Overflow Q & A site for source code comment recommendation. Experiments with 292 Stack Overflow code segments and 5,039 discussion comments show that our approach has a promising recall of 85.42%. We also conducted a complementary user study which confirms the accuracy and usefulness of the recommended comments.**

***Index Terms*—Stack Overflow, code examples, program analysis, code insight, comment recommendation**

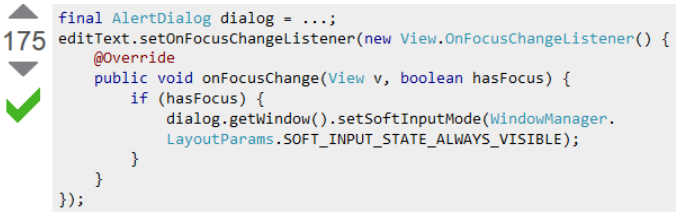
I. Introduction

Studies show that software maintenance can take up to 85%–90% of the total cost of a software product [11, 12]. The most time-consuming task during maintenance is program comprehension, and developers spend about 50% of their time for comprehending the code to be maintained [27]. Despite several studies [27] demonstrating the utility of code comments in comprehension, a few software projects document their code that reduces the future maintenance costs [29]. Self-documentation of the code (e.g., descriptive identifier names) might also occasionally lead to long identifier names which reduce the readability of the code [16]. Thus, one way to help the developers comprehend the code is to automatically generate meaningful code comments that explain the intent of the code, and there exist several techniques for this [18, 29, 30, 32]. However, for gaining a deeper understanding of the code beyond its functionality, the developers may find neither the simple code comments (manually written or auto-generated) nor the descriptive identifier names sufficient enough.

Existing techniques generate comments for different granularities of code such as method [18, 29], code segment [30, 32], and API method call [31]. Sridhara et al. [29] propose a technique that generates the leading comment for a Java method by analyzing both method signature and identifier names used in the method body. Wong et al. [32] propose another code comment generation technique that mines developer's description of the posted code in Q & A site answer as the comment for a similar code fragment. Code comments generated by existing approaches describe what the code does. However, there are other important aspects of the code such as its quality (e.g., maintainability) or issues (e.g., bugs), which are ignored by earlier approaches. In order to reach an actionable understanding (i.e., for code reuse or change task), one needs to further analyze the code which is often a non-trivial task especially for the novice developers. Code comments that provide insight into the quality, issues or scopes for further improvement of the source code are likely to help the developers in this regard.

In this study, we propose an approach that exploits available information on Stack Overflow to provide such additional comments. In Stack Overflow, a posted code example (e.g., Fig. 1) is often followed by a series of conversations (e.g., Fig. 2) among the participants. Such discussion often contains useful insight that can aid a developer in analyzing a piece of code of interest. For instance, the code example in Fig. 1 is posted on Stack Overflow as an answer to the question– *"Android: How to show soft keyboard automatically when focus is on an EditText?"* The code example suggests an `AlertDialog`-based technique for showing the keyboard. The answer is widely viewed (i.e., 131,000 views), recognized by the community (i.e., *score*: 175), and accepted as the *solution* by the person who initially posted the question. The community also posts a number of concerns, tips and observations about the code in the form of comments. Apparently, one could think of these comments as the assertions based on entirely subjective viewpoints. However, such comments are constantly reviewed by other members from the community. More importantly, the Linus's law about software bugs– *"Given enough eyeballs, all bugs are shallow*[1]*"* also perfectly applies to Stack Overflow. Thus, such a discussion often lends itself to a high quality analysis of the posted code. For example, the seventh comment (i.e., marked in green, DC_4, Fig. 2) in the discussion suggests an important *troubleshooting tip* associated with the posted code (Fig. 1) which is recognized by at least 23 other members from the community. Similarly, the eleventh comment (i.e., marked in green, DC_5, Fig. 2) points out an interesting concern about the code for a use case scenario, which is up-voted by at least ten other members. Such insightful information might be invaluable in the static analysis targeting bug fixation or other maintenance of a similar code segment (e.g., Listing 1) from any software

[1] https://www.princeton.edu/∼achaney/tmve/wiki100k/docs/Linus_s_Law.html



```
final AlertDialog dialog = ...;
editText.setOnFocusChangeListener(new View.OnFocusChangeListener() {
    @Override
    public void onFocusChange(View v, boolean hasFocus) {
        if (hasFocus) {
            dialog.getWindow().setSoftInputMode(WindowManager.
            LayoutParams.SOFT_INPUT_STATE_ALWAYS_VISIBLE);
        }
    }
});
```

Fig. 1. Stack Overflow code example (taken from [1])

Fig. 2. Review comments for code example in Fig. 1

project (e.g., `openmidaas-android-app`). Unfortunately, neither manually written comments by developers nor auto-generated comments provide such information.

An automated technique for mining code comments from such resource faces a major challenge. The discussion often contains a number of trivial comments (e.g., marked in red in Fig. 2). The challenge is to automatically separate the useful comments (e.g., marked in green in Fig. 2) from the trivial ones, where the useful comments provide insights on the quality (e.g., maintainability) or issues of the code. Our research attempts to overcome that challenge, mines insightful comments from Stack Overflow discussions, and recommends them as code comments for a given code segment.

In this paper, we propose a heuristic-based technique for mining insightful comments (i.e., discussing issues, concerns or tips) from Stack Overflow for a code segment to be comprehended or analyzed. We first perform an exploratory study that analyzes Stack Overflow discussions, and explores the potentials of Stack Overflow for such insightful comment recommendation. Based on our findings from that study, we (1) collect five heuristics–*popularity, relevance, comment rank, word count* and *sentiment* (i.e., polarity) associated with each of the posted comments for an Stack Overflow code example,

```
final AlertDialog alertDialog = alert.create();
etName.setOnFocusChangeListener(new OnFocusChangeListener()
     {
    @Override
    public void onFocusChange(View arg0, boolean hasFocus)
         {
      if (hasFocus) {
        alertDialog.getWindow().setSoftInputMode(
                WindowManager.LayoutParams.
                     SOFT_INPUT_STATE_ALWAYS_VISIBLE);
      }
    }

  });
  alertDialog.show();
```

Listing 1. Code segment of interest (Line 130–141 from DialogUtils.java [3])

(2) combine their heuristics carefully for ranking, and (3) then identify the *top ranked* comments as the insightful comments. In contrast with Wong et al. [32], that uses the description attached to Stack Overflow code segment as comment, we explore the possibility of using follow-up discussions on Stack Overflow for insightful code comment recommendation.

An exploratory study using 9,016 questions, their accepted answers and 706 popular discussion comments from Stack Overflow shows that about 22% of the comments are useful. They contain meaningful insights on *quality, issues* and *scopes* for further improvement of the posted code which can be leveraged for automatic code comment recommendation. We evaluate our technique in two ways. Experiments using 292 Stack Overflow code segments from *Android, Java* and *C#* domains and 5,039 discussion comments show that our mining technique has a promising recall of 85.42% on average. The user study involving professional developers with 85 open source code segments show that about 80% of the recommended comments by our approach for those segments are found to be *accurate, precise* and *useful* by the participants. Thus, we make the following contributions:

- We conduct an exploratory study that demonstrates the potential of Stack Overflow for insightful code comment recommendation.
- We propose a heuristic-based technique for mining insightful code comments (i.e., discussing issues, concerns, tips) from Stack Overflow for a code segment of interest.

II. Background

A. Topic Modeling

Topic modeling is a statistical modeling technique that retrieves underlying topic-structures of a given text corpus without the need for tags, training data or predefined taxonomies [6, 7]. It uses word frequencies and co-occurrence frequencies in the documents to develop a model of related words. Topic modeling has been successfully used in information retrieval, software engineering and even in computer vision [5]. In software engineering, it is frequently used for topic analysis of the discussions (e.g., mailing list) among developers [6, 10, 13], software evolution analysis [9, 14], bug topic analysis [17], and even for developers' expertise analysis. In this paper, we use topic modeling in order to explore API topic-structures in different Stack Overflow code examples. We

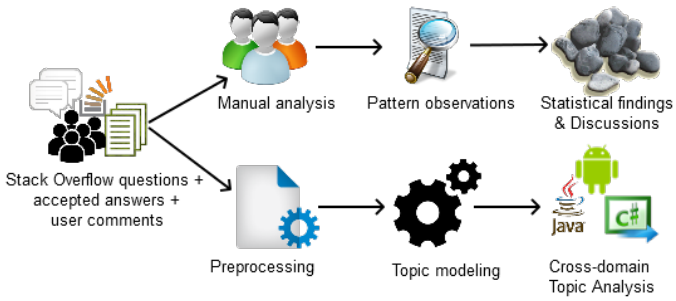


Fig. 3. The overview of the exploratory study

use the topic modeling technique– Latent Dirichlet Allocation (LDA) [7] for our study. LDA is a statistical topic model which considers a document as a mixture of the retrieved topics, and considers a topic as a set of co-occurring words throughout different documents within the corpus.

B. PageRank Algorithm

PageRank algorithm by Lawrence Page and Sergey Brin is an efficient approach for ranking a list of web pages that are inter-linked with one another [8]. It is also widely used in other fields of research such as web spam detection, text mining, text summarization, word sense disambiguation, and natural language processing [4]. The algorithm treats a hyper link in a web page to another website as a vote cast for that site, and it analyzes both incoming links and outgoing links of the page for the ranking. If the web page is highly hyper-linked (i.e., recommended) by other popular pages, it is also considered as a popular page, and vice versa. Thus, the *PageRank* score of the web page can be calculated as follows:

$$PR(A) = (1-d) + d(\frac{PR(T_1)}{C(T_1)} + ... + \frac{PR(T_n)}{C(T_n)}) \quad (1)$$

$PR(A)$ represents the *PageRank* of page A, $PR(T_i)$ represents the *PageRank* of pages T_i that link to page A, d refers to damping factor[2] that has a value between zero and one, and $C(T_i)$ is the number of outbound links on page T_i.

In this research, we adapt this algorithm for *interaction network* of Stack Overflow comments posted for a code example, where each node denotes a comment and each edge denotes interaction between two comments. We exploit user reference relationship (e.g., user *tidbeck* refers to user *Stephen* in the fourth comment at Fig. 2) and posting sequence of comments (e.g., $DC1 \rightarrow DC2 \rightarrow DC3$ at Fig. 2) for developing the comment interaction network.

III. Exploratory Study

One of the primary goals of this research is to find out whether Stack Overflow discussions (i.e., crowdsourced knowledge) are a meaningful source for insightful comments about code, and whether such discussions can be recommended as code comments. We thus manually analyze 706 comments from Stack Overflow discussions in this exploratory study. We then analyze the Stack Overflow code examples that encourage those discussions. Fig. 3 shows the schematic diagram of our study using Stack Overflow questions, their accepted answers, and discussion comments, where we perform manual analysis and cross-domain API topic agreement analysis for answering the following research questions:

- **Exp-RQ$_1$:** Do the follow-up discussions from Stack Overflow contain any useful information (e.g., insight on quality or issues of code) that is likely to aid software maintenance activities such as bug fixation or further code quality improvement?
- **Exp-RQ$_2$:** Which API classes and methods are used in Stack Overflow code examples that encourage those insightful discussions?

TABLE I
Dataset for Manual Analysis

Items	Java	Android	C#	Total
Questions	98	81	103	282
Accepted answers	98	81	103	282
Code segments	101	83	108	292
Discussion comments	276	161	269	706

TABLE II
Dataset for API Topic Analysis

Items	Java	Android	C#	Total
Questions	2,932	2,657	3,427	9,016
Code segments	2,544	2,438	3,208	8,190

A. Data Collection

We collect a total of 706 comments from the discussions of 282 accepted Stack Overflow answers from three domains– *Java, Android* and *C#* using Stack Exchange Data API[3]. Since we manually analyze the comments for meaningful information, we choose each of them under certain restrictions– (1) the Stack Overflow answer for which the comment is posted should contain at least ten lines of code, in order to ensure that the answer is mostly about programming, and (2) each chosen comment should be voted by at least five Stack Overflow users, in order to avoid less important comments. It should be noted that Stack Overflow contains millions of comments, and manually analyzing them all is impractical. We thus collect a popular and non-trivial subset of 706 comments from a total of 5,039 comments from 282 Stack Overflow answers, and Table I shows the dataset for our manual analysis.

We also collect a set of 9,016 Stack Overflow questions and their accepted answers (Table II) for the study from the three domains. We extract code segments from the natural language text of answers using *Jsoup*, a popular HTML parser. Given that we are interested to analyze such code segments that encourage follow-up discussions, we chose the questions and their accepted answers under certain restrictions– (1) each of the questions must be viewed at least 500 times to ensure that the question is widely viewed by the community, (2) the answer must contain one or more code segments as a part of the solution, where each segment contains at least three lines of code, and (3) the answer must have at least ten comments by users. In Stack Overflow, code segments are generally posted using `<code>` tags [25], and we extract the inner text of

[2]The probability of jumping from one page to another by a visitor

[3]http://data.stackexchange.com/stackoverflow/queries

those tags as code segments. We also automatically check the code segments using several heuristics, and discard 6%-13% segments as false positives from each domain, that leaves us with a total of 8,190 segments. False positives mainly occur due to wrong content (i.e., other than code) inside those tags.

B. Answering Exp-RQ$_1$: Analysis of Discussion Comments

To answer Exp-RQ$_1$, we analyze 706 discussion comments that are extracted during the data collection step (Section III-A). In Stack Overflow, each discussion for an answer takes the form of a series of comments by the users, where the comments mostly focus on the posted code segments in the answer. We analyze such review comments manually, and determine if they contain any meaningful information that might assist in the comprehension or other maintenance activities such as bug fixation or further quality improvement of a code segment of interest. Our manual analysis is divided into the following sections:

1) Classification of Comments: We manually analyze 706 chosen discussion comments and their corresponding code segments, answers and questions, where we attempt to determine the intent behind the comments. We particularly identify if the comments contain any mention of possible bugs (i.e., does not work for certain cases) or limitations (e.g., lack of portability, readability or security) that the code might face or if they contain any useful tips for improving the code. In Stack Overflow, users from different levels of expertise take part in the discussions, and such comments are often posted by them based on their work experience. Thus, the posted information is likely to assist one in troubleshooting or reusing that code. We identified several intents behind those comments, and categorize them into seven categories as follows:

Clarification Question ($\mathbf{C_1}$): There exist several comments that request for more information or for confirmation about certain observations on the posted code, and finish with question marks. We categorize them as *clarification questions*. For example, in Stack Overflow, the comment (ID: 6213963):

> *"Don't you miss a return when drawableMap contains the image ... without starting the fetching-thread?"*

is posted against a code segment that loads images in Android `ListView`. The comment identifies an important concern and requests for feedback.

Code Documentation ($\mathbf{C_2}$): The comments that explain how a posted code segment works are categorized as *code documentations*. We note that some of the discussion comments can also act as a documentation for the code segment which the discussion follows. For example, the comment (ID: 7867648):

> *"... look into how Intents work to understand this. It'll basically open an email application with the recipient, subject, and body already filled out. It's up to the email app to do the sending."*

is posted by the answerer to explain a code segment that uses `Intent` for sending email from an Android application.

Tips & Complementary Information ($\mathbf{C_3}$): Given that a number of users with varying expertise view a posted answer and especially the code segments posted within it, their observations or suggestions on the code are worth noticing. Sometimes, those comments discuss useful tips (e.g., workaround for certain issues) for the code or complementary information for further analysis and improvement of the code. For example, in Stack Overflow, the comment (ID: 11919416):

> *"... as you do the mReceiver.setReceiver(null); in the onPause method, you should do the mReceiver.setReceiver(this); in the onResume method. Else you might not receive the events if your activity is resumed without being re-created."*

is posted against the code segments that handle REST API calls within an Android application, and the comment provides a useful tip for further improvement of the code.

Bug, Concern & Limitation ($\mathbf{C_4}$): The Linus's law regarding software bugs also applies for the code segments posted on Stack Overflow. Given that a large crowd of technical users is involved in viewing and reviewing the posted answers or their code, different aspects of a code segment are explored and the discussion often points out possible bugs, errors, warnings or important concerns associated with the code. Such information is of utmost importance to other users (e.g., developers) troubleshooting or analyzing that code. For example, one user posted the comment (ID: 10747399):

> *"This code does not work properly. Some '0' characters becomes missing in the generated string. I don't know why, but that's the case."*

after reviewing an Android code segment that encrypts a string, and the comment clearly reports a bug within the code.

Strength Statement ($\mathbf{C_5}$): These comments generally contain praising words about the posted code, its elegance or strength, and how it does help others in their works. For example, this comment (ID: 149203):

> *"Best damn piece of code I have seen :), Just solved a million problems in my project :)"*

is posted for a code segment that deals with dynamic `LINQ` and `IEnumerable` in C#. These comments also often contain trivial thank statements.

Miscellaneous ($\mathbf{C_6}$): There exist comments whose intentions cannot be easily explained. They do not fall into any of the above categories. We call them *miscellaneous* comments. During our analysis, we found a major part of the discussion comments belong to this genre. For example, the comment (ID: 2903068):

> *"You should really cite your references. This algorithm was invented by Robert Floyd in the '60s, It's known as Floyd's cycle-finding algorithm, aka. The Tortoise and Hare Algorithm."*

is posted against a code segment that partially implements *Floyd's cycle-finding algorithm* for detecting loops in linked list. The comment emphasizes on the citation of the algorithm used in the code. Other miscellaneous comments express

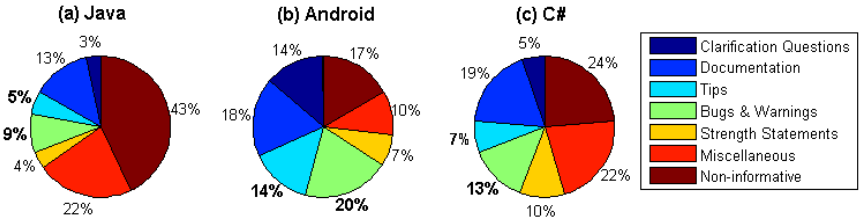


Fig. 4. Discussion comment statistics

users' frustration about particular technology or programming language, social humour and technology history.

Non-informative Comments ($\mathbf{C_7}$): During manual analysis, we also note that a significant part of our chosen comments is not related to code although the answers contain code segments. In Stack Overflow, users sometimes post code segments in order to answer different conceptual questions where source code might not be an essential part. In such cases, the follow-up discussion might focus on the overall answer rather than the code segments. We discard such comments from our further analysis, and categorize them as *non-informative* comments.

2) Statistics for Comment Classes: We manually label each of the 706 comments into seven categories, which are later used as the *gold comments* [2] for evaluation (Section V). Fig. 4 shows the fraction of different categories of popular discussion comments that follow the code segments or programming answers from the three domains- *Java, Android* and *C#*. We are particularly interested about these two categories– *bugs* (C_4) and *tips* (C_3) as they contain non-trivial information and the users making such comments are likely to possess a certain level of expertise on the posted code. We observe that each of the programming domains contains a significant amount of such comments which is promising. For example, 20% of *C#* comments and 14% for *Java* comments belong to these categories. On the other hand, *Android* has the maximum of 34% comments that contain identified *bugs* and improvement or troubleshooting *tips* for the posted code. One possible explanation could be that *Android* is relatively less mature than the other two platforms, and thus, it encourages more discussions about bugs and further improvement. Such findings actually underpin our intuition about Stack Overflow discussion comments. Although the comment classification is a bit subjective (i.e., might face reproducibility issue), and the dataset analyzed is not large, the findings are still convincing enough to demonstrate that Stack Overflow comments are a potential source for insightful information about the code.

Our analysis in Exp-RQ$_1$ shows that a significant fraction of the comments posted on Stack Overflow contain useful (i.e., insightful) information, and such comments are likely to aid different software maintenance activities such as bug fixation or code quality improvement.

C. Answering Exp-RQ$_2$: Automated API Analysis

Since our manual analysis suggests that Stack Overflow discussions contain meaningful information for software maintenance tasks, we are interested to investigate the program artifacts in the answers that mostly encourage those discussions. To answer Exp-RQ$_2$, in this section, we analyze the code segments from Stack Overflow answer texts using topic modeling, and perform cross-domain topic-agreement analysis among three domains– *Java, Android* and *C#*. Our semi-automated analysis has the following steps:

1) Preprocessing: In this step, we analyze each of the code segments from the texts of 9,016 Stack Overflow answers (Table II), where we extract different API elements (e.g., *class name, method name*) from the code. We decompose dotted tokens (e.g., `java.util.HashMap`) into individual tokens, and exploit *camel-case notations* for extracting the API elements. It should be noted that we do not include the frequent programming keywords (e.g., `if,for`) or punctuation tokens (e.g., `([?;)`) in the list. Thus, the *API corpus* of each of the three programming domains contains a list of files containing API class and method names used in the corresponding accepted answers from that domain.

2) Performing Topic Modeling: In order to identify API topic trends in the code segments from each programming domain, we perform topic modeling on the corresponding *API corpus*. We make use of *Mallet* [19], an LDA-based popular topic modeling tool, for the task. We extract a maximum of 150 API topics given that a large number for topics, K, helps identify specified or recognizable topics [9]. We thus chose the hyper parameters– α, β for the model carefully. We consider a symmetric version of α, probability distribution of topics over the corpus, where all topic distributions sum to one and each distribution equals to $1/K$. A lower value of α indicates that the model assumes that each document in the corpus focuses on a few number of topics (i.e., one or two) [7]. We also chose a lower value of 0.006 for β, distribution of topics over words from the corpus. A lower value of β indicates that the model assumes that each of the topics can be represented using a few words [6, 7]. In our case, the tool automatically optimizes the topic words, and each of the 150 topics is represented using only six words. We extract such 150 API topics for each of the three programming domains under study.

3) Cross-domain API Topic Analysis: Once topics are extracted, we filter out the topics with a β value less than our *adopted threshold* (i.e., 0.006) which leaves us with 26 Java API topics, 43 Android API topics, and 44 C# API topics. We consider the top five API topics for each of the documents in the corpus, determine *topic-document-frequency*, and sort the topics according to their frequencies. Due to space limitation, the ranked topics are available for download elsewhere[4]. We then investigate if there exists any regular pattern among the topics from different domains. We first assign a label to each

[4]http://www.usask.ca/∼mor543/codeinsight/topics

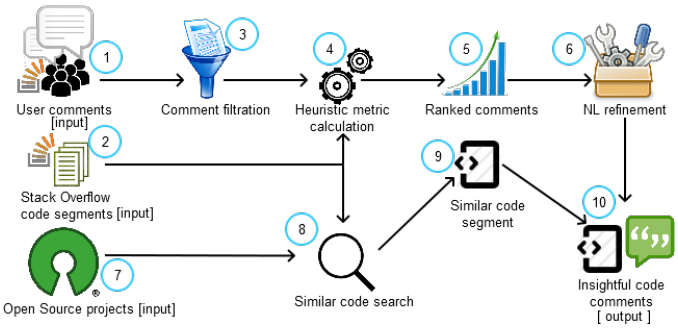


Fig. 5. Schematic diagram of the proposed technique

of the topics by analyzing the API names and occasionally the associated code examples [14]. We interestingly notice that several topic labels across the domains are semantically related, and we thus group them into a single category. Finally, we found five such categories– (1) *List & Collections*, (2) *I/O Operations*, (3) *String Operations*, (4) *Date & Time*, and (5) *Integer*. We found that among those five categories, *Collections* and *I/O* APIs are commonly used in the code segments across all three domains that encourage follow-up discussions. The other two API categories associated with string operations, date and time are dominant in *Java* and *C#* domains but not in Android domain. We also found that the API for parsing integer numbers is dominant in the code segments that attract discussions especially in *Java* and *Android* domains. Such findings are particularly important since they partially explain the characteristics of the artifact (i.e., API) that might trigger the discussions of our interest at Stack Overflow. Besides, they might benefit API-specific comment mining techniques towards improved API documentation.

Our analysis in Exp-RQ$_2$ shows that several APIs are used in the code segments that encourage follow-up discussions at Stack Overflow. However, five of them are not only frequent but also common across the three domains under study.

IV. CodeInsight: Mining Insightful Code Comments for Source Code

Fig. 5 shows the schematic diagram of our proposed technique for mining insightful comments from Stack Overflow for a given code segment. The exploratory study (Section III-B) analyzes potential of discussion comments from Stack Overflow, and suggests a classification for them that helps us focus on certain comment categories. Since we are interested about the insightful code comments discussing bugs or improvement tips for code, a technique is required for automatically mining the discussion comments of corresponding categories–C_4 (i.e., *bugs*) and C_3 (i.e., *tips*). We discard insignificant (i.e., non-popular) comments from the discussion using *vote count threshold* (Step 3, Fig. 5), apply a set of five heuristics for ranking, and then extract the top-ranked comments (Step 4, 5). Finally, we refine the extracted comments using natural language processing tools (Step 6) so that they can be recommended as the code comments for a similar code segment (Step 7, 8, 9) to be analyzed for maintenance.

A. Proposed Heuristics

Since our preliminary investigation shows that the type of comments (e.g., *bugs (C4)*) generally does not correlate with its linguistic characteristics such as *sentence count* or *readability*, we carefully choose and apply a list of five heuristics for mining the important comments. While *popularity* and *relevance* of a candidate comment with respect to the posted code are important aspects, we also heuristically capture *comment size*, *impact* of that comment in the discussion and the *sentiment* (e.g., positive or negative) expressed in the comment texts for identifying insightful comments as follows:

Popularity (P): In Stack Overflow, comments in the follow-up discussion for a posted answer are often subjectively evaluated by other users. If a comment adds something useful to the answer, it gets up-votes, otherwise, it is marked as a trivial or non-productive comment. In our research, we consider *popularity* of comment as a heuristic for comment ranking, and we use *up-vote count* as an estimate of its *popularity* within the discussion.

Word Count (WC): During manual analysis, we note that follow-up discussions in Stack Overflow might contain several comments that are too short to contain any meaningful information. For example, the showcase discussion (Fig. 2) contain three such comments marked in red. We thus consider *word count* as a heuristic for comment ranking, and discard such trivial comments during ranking.

Relevance (R): In a follow-up discussion, there might be several comments which either apply to several code segments or do not refer to any of the code segments at all from the posted answer. These comments are not suitable for our purpose, and thus we consider *relevance* of comment as another important heuristic for comment ranking. In the discussion comments, Stack Overflow users often refer to different code elements such as *class names, method names* or *package names* or different programming concepts (e.g., list, sorting) from the posted code segments of an answer. Thus lexical similarity between the code and a candidate comment is a potential estimate for relevance of that comment to the code. We use *cosine similarity* measure for determining the lexical similarity [25], where we apply different preprocessing (e.g., stop word and punctuation removal, word splitting) both on the code segments and the comments. It should be noted that we avoid stemming for both items due to their heterogeneous nature (e.g., code and natural language) of textual content.

Comment Rank (CR): In the discussion comments, Stack Overflow users sometimes refer to other users participating in the discussion using *"@user"* notation. For instance, in showcase example (Fig. 2), user *tidbeck* refers to user *Stephen* using word– *@Stephen* at the beginning of the fifth (i.e., DC_3) comment. Such reference indicates that not only this comment is a response to the previous comment (i.e., first comment, DC_1) posted by *Stephen* but also the previous comment is a stimulus for the discussion. We identify such stimulating and stimulated comments in the discussion, and develop a *comment interaction network*. We also use the sequence

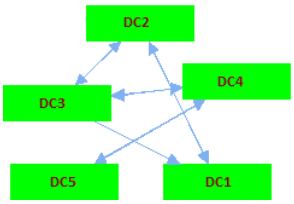


Fig. 6. Comment interaction network in Fig. 2

of comments based on their *comment id* in developing the interaction network, and in this case, the baseline heuristic is that a comment not only does get influenced by the immediate previous comment in the conversation but also it influences the following comment. For example, Fig. 6 shows the interaction network for the comments (i.e., filtered by *vote count*) in Fig. 2, where each node refers to a distinct comment (i.e., labeled with comment ID) and each edge denotes an interaction between the two comments. We apply *PageRank algorithm* (Section II-B) on this network to recursively calculate the *score* for each node by analyzing not only its connectivity in the network but also the scores of the connected nodes. The algorithm runs until the scores for each of the nodes converge, and such a score can be considered as an estimate of relative importance of a node in the network. We consider that score as *comment rank* (i.e., analogous to *page rank* for web pages [8]), and apply it as a heuristic for comment ranking.

Sentiment (S): From the exploratory study (Section III-B), we notice that discussion comments that point out possible bugs and warnings (C_4) or troubleshooting tips (C_3) for the posted code are often associated with negative emotions such as frustration, annoyance and disappointment. On the other hand, comments that contain appreciation about the posted code emit positive emotions such as gratitude or thankfulness. Since we are interested in mining the comments containing bugs, concerns and tips for improving the code, sentiment expressed in the comment texts is a potential heuristic for separating them from rest of the comments in the discussion [21]. We use a state-of-the-art sentiment analysis tool– *Stanford Sentiment Analyzer* [28] for determining sentiment of each of the comments. The tool returns a sentiment score on the scale from "0" (i.e., very negative) to "4" (i.e., very positive) for a sentence, and we adapt the scale so that it ranges from "-2" (i.e., very negative) to "+2" (i.e., very positive) and "0" denotes neutral sentiment. We parse each of the sentences from a candidate comment, sum up their sentiment scores, and then use it for the ranking of the discussion comments.

B. Heuristic-based Comment Ranking & Selection

Once the discussion comments are collected from Stack Overflow, less important (i.e., non-popular) comments are filtered out. In particular, we solve the above problem by considering only such comments which have scored at least one up-vote to date and discarding the rest from further analysis. This filtering step helps us move on with a moderate list containing relatively popular and relatively important comments. We then collect each of the five heuristics– *popularity, relevance, comment rank, word count* and *sentiment* for those comments, and develop separate ranked lists for each of the heuristics. It should be noted that the discussion comments are ranked in ascending order for sentiment heuristic (i.e., negative sentiment associates with bugs) whereas they are ranked in the opposite order for the rest four heuristics.

Since each individual heuristic captures a distinct aspect of salience for the comments, top comments in each of those five rankings are potential candidates for final selection. We choose *top five* comments from each ranking, and determine the frequency for each of the comments in those top ranks (adapted from Holmes and Murphy [15]). We then choose the top $K = 3$ comments based on their frequencies as the insightful comments. Such ranking and selection of comments ensure that the extracted comments are not only *popular* among the users but also *non-trivial*, *stimulating* for discussion and *relevant* to the posted code. More interestingly, they point out important issues or concerns in the code in terms of negative *sentiment*. For example, our heuristic algorithm extracts DC_4, DC_5 and DC_1 from the discussion in Fig. 2 as the insightful comments, where the first comment, DC_4, provides an important tip for troubleshooting, and the second one, DC_5, reports an important concern about the posted code.

C. Comment Text Refinement

Discussion comments from Stack Overflow often contain personal pronouns (e.g., *"I", "you",* etc.) and possessive pronouns (e.g., *"mine", "yours",* etc.). As suggested by existing studies [32], such words contribute no value in a comment for code, and we thus replace them with more objective words. For example, the comment–*"You should not use the IV like this. For a given two messages, they should not have been encrypted with the same Key and same IV"* is transformed into *"One should not use the IV like this. For a given two messages, they should not have been encrypted with the same Key and same IV"*. We use Stanford CoreNLP POS tagging [28] for identifying such pronouns, and replace them selectively in order to make the comments formal. We also remove user reference information (e.g., *@Stephen*), replace numbers (e.g., *"3"*) with numerical words (e.g., *"three"*), and transform the spoken words (e.g., *"can't", "doesn't"*) into formal words (e.g., *"can not","does not"*) to make the comments useful.

V. Evaluation of CodeInsight

We conduct two experiments– an empirical study and a user study for evaluating our proposed technique for insightful code comment generation from Stack Overflow for a given code segment. The goal is to evaluate both the performance of the comment ranking algorithm and the quality of the recommended comments [32]. We thus attempt to answer the following three research questions using those experiments:

- **RQ1**: How effective the proposed technique is in retrieving the comments that discuss bugs, concerns and tips for improvement in the posted code?
- **RQ2**: Are the recommended comments *accurate, precise* and *concise* in describing the potential issues or troubleshooting tips for the target code segment?

TABLE III
EXPERIMENTAL RESULTS FOR DIFFERENT HEURISTICS

Heuristics Used	Metric	Java		Android		C#		Average	
		C_3[1]	C_4[2]	C_3	C_4	C_3	C_4	C_3	C_4
{Popularity (P)}	CE[3]	09(15)	16(24)	22(23)	28(32)	17(19)	25(36)	–	–
	Recall	60.00%	66.67%	95.65%	87.50%	89.47%	69.44%	81.71%	74.53%
	MRR[4]	0.44	**0.50**	0.59	**0.57**	0.47	**0.56**	0.50	**0.54**
{Popularity (P), Relevance (R)}	CE	09(15)	16(24)	22(23)	28(32)	16(19)	29(36)	–	–
	Recall	60.00%	66.67%	95.65%	87.50%	84.21%	80.56%	79.95%	78.24%
	MRR	0.44	0.44	0.64	0.54	0.50	0.48	0.53	0.49
{Popularity (P), Relevance (R), Comment Rank (CR)}	CE	10(15)	17(24)	22(23)	27(32)	16(19)	28(36)	–	–
	Recall	**66.67**%	70.83%	95.65%	84.38%	84.21%	77.78%	82.18%	77.66%
	MRR	**0.60**	0.29	**0.73**	0.56	0.44	0.50	**0.59**	0.45
{Popularity (P), Relevance (R), Comment Rank (CR), Word Count (WC)}	CE	07(15)	18(24)	22(23)	28(32)	17(19)	30(36)	–	–
	Recall	46.67%	75.00%	95.65%	87.50%	89.47%	83.33%	77.26%	81.94%
	MRR	0.57	0.33	0.59	0.50	**0.53**	0.47	0.56	0.43
{Popularity (P), Comment Rank (CR), Relevance (R), Word Count (WC), Sentiment (S)}	CE	09(15)	19(24)	22(23)	31(32)	18(19)	31(36)	–	–
	Recall	60.00%	**79.16**%	**95.65**%	**96.88**%	**94.74**%	**86.11**%	**83.46**%	**87.38**%
	MRR	0.44	0.32	0.55	0.52	0.33	0.45	0.44	0.43

[1]Comments containing improvement tips, [2]Comments containing bugs and warnings, [3]Comments retrieved, [4]Mean Reciprocal Rank

- **RQ3**: Are the recommended comments *useful* for static analysis involving maintenance of the target code?

A. Evaluation of Comment Ranking Algorithm

In the first experiment, we evaluate our ranking algorithm that ranks a list of discussion comments from Stack Overflow based on the proposed heuristics (Section IV-A), and recommends the three most insightful comments for a code segment to be analyzed or comprehended. We use 5,039 discussion comments targeting 292 code segments, and 706 gold comments (i.e., manually labeled) (Table I) from three popular domains– *Java, Android* and *C#* for this experiment. Qualitative analysis on the gold comments [2] can be found in Section III-B. Our algorithm successfully retrieves 130 out of 149 comments (C_3 and C_4) discussing *bugs, warnings* and *tips* about the posted code from the total of 5,039 Stack Overflow comments. We use two performance metrics–*recall* and *mean reciprocal rank* (MRR) for the evaluation. *Recall* denotes the fraction of the gold comments that is retrieved by a technique whereas *reciprocal rank* refers to the multiplicative inverse of the first relevant item's position in the ranked list. *Mean reciprocal rank* averages such measure for all trials. It should be noted that precision is not calculated as a performance metric since we recommend only three comments at once.

From Table III, we note how each of the five heuristics– *popularity, relevance, comment rank, word count* and *sentiment* contributes to the final ranking and selection of the discussion comments. We incrementally add different heuristics to the ranking algorithm, and analyze their performance for avoiding any potential suboptimal set of heuristics. We note that *popularity* heuristic dominates others to a large extent, and this occurs probably due to our design choice for the dataset. We exploit mass evaluation for the comments by a large technical crowd in order to avoid subjective bias, and chose a popular (i.e., up-voted) subset of all comments for manual analysis (Section III-B) and experiment. Although the *popularity* heuristic performs considerably well in mining important *tips* (i.e., *recall* 81.71%), it does not perform equally (i.e., *recall* 74.53%) for *bugs* and *warnings*. We thus add more heuristics gradually, and achieved a global maximum in terms of *recall* for both comment types when all five heuristics are combined. It should be noted that *mean reciprocal ranks* do not show similar behaviour, and they mostly remain comparable. In short, our comment ranking algorithm retrieves C_3 comments (e.g., *tips*) and C_4 comments (e.g., *issues* or *concerns*) with a promising *recall* of 85.42% on average. Although the dataset (i.e., 5K comments) used is not large, the findings clearly demonstrate the potential of our ranking technique. Such findings also answer our first research question, RQ_1.

B. Evaluation of Recommended Comments: A User Study

Although the empirical evaluation clearly shows the high potential of our proposed technique, we also wanted to see whether the developers do like the technique and find the recommended comments useful. This leads to our second evaluation, and we conducted a user study involving four professional developers as well. In collaborative software development, software developers often comprehend or analyze the code written by peers in the form of code reviews or code reuse. We attempt to determine if the professional developers, i.e., study participants, find our recommended comments *accurate, precise, concise* or *useful* for analyzing the code segments that are similar to Stack Overflow code segments but are taken from open source projects [32].

Dataset Preparation for User Study: We conduct a case study with 82 open source projects for collecting dataset for the user study. Our exploratory study (Section III-B) and empirical evaluation (Section V-A) involve 292 Stack Overflow code segments, and in this case, we locate similar segments in open source projects. We exploit *GitHub code search* for finding similar code since it is a preferable choice for our task. GitHub hosts a large collection of open source projects, and its *code search feature* extracts results from hundreds of projects. Thus, it has a greater chance of retrieving the code segments of interest, and existing studies [24] also show such findings. We also found that code search using appropriate keywords followed by a careful manual analysis of the top results can actually locate the code segments of interest.

We perform an exhaustive search using GitHub search, and locate 85 code segments similar to Stack Overflow code

TABLE IV
DEVELOPER JUDGEMENT ON RECOMMENDED COMMENTS

	Java				Android				C#				Average			
Responses	**Ac**	**Pr**	**Co**	**Us**	**Ac**	**Pr**	**Co**	**Us**	**Ac**	**Pr**	**Co**	**Us**	**Ac**	**Pr**	**Co**	**Us**
Strongly Agree	13	11	8	12	20	16	16	18	11	10	12	11	82.50%	80.83%	78.33%	79.17%
Agree	4	7	8	5	11	11	12	13	6	7	5	4				
Neutral	2	0	3	1	2	7	7	4	1	1	1	3	6.67%	7.50%	12.50%	10.00%
Disagree	0	1	0	1	4	4	4	3	2	2	2	2	10.83%	10.00%	9.17%	10.83%
Strongly Disagree	1	1	1	1	3	2	1	2	0	0	0	0				
Total	20	20	20	20	40	40	40	40	20	20	20	20				

Ac=Accurate, **Pr**=Precise, **Co**=Concise, and **Us**=Useful

segments in the open source projects. We found that most of them are either directly copied (i.e., Type 1 clone) or slightly modified (i.e., Type 2 clone) before posting to Stack Overflow. Thus, they are suitable candidates for the user study that evaluates the quality of the comments from Stack Overflow recommended for similar code segments.

Study Participants: In our user study, we involve four professional developers from two reputed software development companies specialized in web technology and mobile applications with 5-10 years of experience. The developers have professional experience on each of the three programming domains that ranges from 1 to 2.5 years for standard *Java*, 1.5–5 years for *Android* and 1.5–3.5 years for *C#*. Such experience makes them suitable candidates for our study.

Study Setup: In the study, each of the participants worked with 20 code segments (i.e., 5 for *Java*, 10 for *Android* and 5 for *C#*) and spent about 2 hours on average. The code segments are randomly chosen from the collection of 85 open source code segments. We provide top three comments by our technique for each of those code segments (along with their technical problem context), and ask the participants to evaluate those comments. The goal is to determine if such comments can provide any meaningful information beyond simple explanation, i.e., can provide insights about quality, issues or tips for further improvement of the code. We ask each of the participants to report their responses about the quality of the comments in terms of *accuracy, preciseness, conciseness* and *usefulness* using a five-point Likert scale. We collect 80 responses against those four quality aspects from each of the participants, and this provides a total of 320 data points for evaluation. Thus, we collect sufficient data points, and also partially handle the threat involving small number of participants. However, adding more participants and using more data points would surely further strengthen the findings.

Study Results & Discussions: Table IV summarizes our findings from the user study. We note that each of the participants found most of the recommended comments *accurate*, *precise*, *concise* and *useful* from each of the three domains. In particular, around 83% of the comments for *Java* and *C#* code segments are found *salient* whereas that statistic is 73.13% for *Android* code segments. We also note that around 7%-14% of the comments are marked as irrelevant or non-informative whereas the participants remained undecided for 7%-12% of the comments for different domains. We thus collect observations and suggestions from each of the participants, and attempt to better explain such findings. First, they encountered a few comments which hint about certain issues in the code, but they were expecting more details (i.e., not *precise*). Second, one of the participants encountered three comments that refer to certain identifier names which were not in the code segment (i.e., identifiers changed before posting). Thus, the participant found those comments *inaccurate*. Third, according to another participant, there were one comment that poses silly arguments which are not helpful for program analysis (i.e., not *useful*). In order to address such subtle issues with Stack Overflow comments, more sophisticated filtration is a possible choice which we consider as a future work.

Thus, according to our conducted user study, above 80% of the recommended comments are found *accurate* and *precise* and about 79% of them are found *concise* and *useful* by the participants. This clearly answers RQ_2 and RQ_3, and suggests high quality of the recommended comments.

VI. THREATS TO VALIDITY

We identify the following threats to the validity of our study. First, the dataset (especially discussion comments) used for empirical evaluation (Section V-A) is limited which someone could consider as a threat. However, such data are carefully extracted from a larger dataset based on certain well-defined restrictions (Section III-A). The goal is to analyze discussion comments associated with code segments, and thus, we discard a major fraction of Stack Overflow comments that are not associated with the code, i.e., accepted answer does not contain a code example. Besides, we had to manually analyze those comments in the exploratory study, and analyzing a large amount of comments is impractical. Thus, the dataset might be sufficient enough to validate our findings.

Second, we only involved four professional developers in the user study. While more participants could have been useful, we should note that the empirical evaluation clearly shows the high potential of the proposed technique and that additional observations with the professional developers only confirm the quality of the recommended comments.

Third, we note that Stack Overflow users often use code segments from open source projects as a part of the posted solutions. However, code segments from legacy or proprietary projects might not be shared at Stack Overflow. Thus, our technique may fall short in recommending comments for the code segments from such projects. However, our findings clearly demonstrate its potential for open source projects.

VII. RELATED WORK

A number of studies from the literature generate comments for different granularities of code such as method [18, 29, 30],

class [20], code segment [30, 32, 33], and API method call [22, 31]. Sridhara et al. generate automatic comments both for an entire Java method [29] and a group of code statements [30]. They exploit semantics and linguistic clues in method signatures and identifier names, and syntactic code blocks (e.g., conditional block, loop block) respectively, and generate equivalent natural language descriptions as the comments. Thus, their techniques are subject to the quality of identifier and method names, and they can generate comments only for a limited set of code structures (e.g., one method body [29], groups of method calls [30]). McBurney and McMillan [18] improve their technique [29] by combining method context (i.e., dependent methods) with method body. Wong et al. [32] generate comments for a code segment of interest by mining developer's description on the similar posted code in Q & A site answers. Since they rely entirely on developer's description, quality of the generated comments is subject to the code documentation experience of the developer. Moreover, no auto-generated comments [18, 29, 30] provide any insight into the *quality, issues* or *scopes* for further improvement of the code. On the other hand, our heuristic-based technique mines various insights from Stack Overflow discussions led by a large technical crowd, refines them using NLP tools, and then recommends as code comments for a given code segment.

There exist other studies related to our work that mine *API method descriptions* [9, 22, 23, 31] and *code elements* (e.g., method call) [26] from developer communication documents such as bug reports and mailing lists, forums and programming Q & A sites. Roehm et al. [27] conduct an observational study, and report that there exists a gap between program comprehension research and actual comprehension in practice. While these studies focus on mining API documentations or comprehension patterns, we mine insightful comments from Stack Overflow for a given code segment that discuss issues, concerns or scopes for improvement in the code. From technical point of view, our approach complements the existing comment generation and API description mining techniques since they simply focus on explaining the code and the API elements respectively.

VIII. Conclusion & Future Work

To summarize, we propose a mining technique that mines insightful comments from Stack Overflow for a given code segment, where the comments reveal identified *issues, deficiencies* and *scopes* for further improvement in the code. We first conduct an exploratory study where we analyze Stack Overflow follow-up discussions, and report that the comments contain useful information for static analysis targeting software maintenance. We then propose a mining technique for automatic comment recommendation. Our approach mines the comments by analyzing five aspects of comments– *popularity, relevance, comment rank, word count* and *sentiment* expressed in the text. Experiments with 292 Stack Overflow code segments and 5,039 discussion comments show that our approach can extract the insightful comments with a promising *recall* of 85.42% and a MRR of 0.44 on average. A user study with professional developers and 85 code segments also show that about 80% of the recommended comments are found *accurate, precise, concise* and *useful* by the participants. In future, we plan to integrate our technique in the IDE.

References

[1] Showcase Stack Overflow Answer. URL http://stackoverflow.com/a/2418314.

[2] CodeInsight. URL http://www.usask.ca/~masud.rahman/codeinsight.

[3] OpenMidaas Library. URL https://github.com/securekey/openmidaas-android-app.

[4] A. Arif, M.M. Rahman, and S.Y. Mukta. Information retrieval by modified term weighting method using random walk model with query term position ranking. In *Proc. ICSPS*, pages 526–530, 2009.

[5] K. Barnard, P. Duygulu, D. Forsyth, N. de Freitas, D. M. Blei, and M. I. Jordan. Matching Words and Pictures. *J. Mach. Learn. Res.*, 3:1107–1135, 2003.

[6] A. Barua, S. W. Thomas, and A. E. Hassan. What are Developers Talking about? An Analysis of Topics and Trends in Stack Overflow. *Empirical Software Engineering*, 19(3):619–654, 2014.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[8] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[9] J.-C. Campbell, Chenlei Zhang, Zhen Xu, A. Hindle, and J. Miller. Deficient Documentation Detection a Methodology to Locate Deficient Project Documentation using Topic Analysis. In *Proc. MSR*, pages 57–60, 2013.

[10] M. Dredze, H.M. Wallach, D. Puller, and F. Pereira. Generating Summary Keywords for Emails Using Topics. In *Proc. IUI*, pages 199–206, 2008.

[11] L. Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2 (3):17–23, 2000.

[12] L. Favre. Modernizing Software & System Engineering Processes. In *Proc. ICSENG*, pages 442–447, 2008.

[13] A. Hindle, M.W. Godfrey, and R.C. Holt. What's Hot and What's Not: Windowed Developer Topic Analysis. In *Proc. ICSM*, pages 339–348, 2009.

[14] A. Hindle, N.A. Ernst, M.W. Godfrey, and J. Mylopoulos. Automated Topic Naming to Support Cross-Project Analysis of Software Maintenance Activities. In *Proc. MSR*, pages 163–172, 2011.

[15] R. Holmes and G. C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proc. ICSE*, pages 117–125, 2005.

[16] B. Liblit, A. Begel, and E. Sweetser. Cognitive Perspectives on the Role of Naming in Computer Programs. In *Proc. PPIG*, 2006.

[17] L. Martie, V.K. Palepu, H. Sajnani, and C. Lopes. Trendy Bugs: Topic Trends in the Android Bug Reports. In *Proc. MSR*, pages 120–123, 2012.

[18] P. W. McBurney and C. McMillan. Automatic Documentation Generation via Source Code Summarization of Method Context. In *Proc. ICPC*, pages 279–290, 2014.

[19] A. K. McCallum. MALLET: A Machine Learning for Language Toolkit, 2002. URL http://mallet.cs.umass.edu.

[20] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proc. ICPC*, pages 23–32, 2013.

[21] N. Novielli, F. Calefato, and F. Lanubile. Towards Discovering the Role of Emotions in Stack Overflow. In *Proc. SSE*, pages 33–36, 2014.

[22] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining Source Code Descriptions from Developer Communications. In *Proc. ICPC*, pages 63–72, 2012.

[23] C. Parnin and C. Treude. Measuring API Documentation on the Web. In *Proc. Web2SE*, pages 25–30, 2011.

[24] M. M. Rahman and C. K. Roy. On the Use of Context in Recommending Exception Handling Code Examples. In *Proc. SCAM*, pages 285–294, 2014.

[25] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE*, pages 194–203, 2014.

[26] P. C. Rigby and M.P. Robillard. Discovering Essential Code Elements in Informal Documentation. In *Proc. ICSE*, pages 832–841, 2013.

[27] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How Do Professional Developers Comprehend Software? In *Proc. ICSE*, pages 255–265, 2012.

[28] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts. Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank. In *Proc. EMNLP*, 2013.

[29] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards Automatically Generating Summary Comments for Java Methods. In *Proc. ASE*, pages 43–52, 2010.

[30] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically Detecting and Describing High Level Actions within Methods. In *Proc. ICSE*, pages 101–110, 2011.

[31] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. CODES: Mining Source Code Descriptions from Developers Discussions. In *Proc. ICPC*, pages 106–109, 2014.

[32] E. Wong, J. Yang, and L. Tan. AutoComment: Mining Question and Answer sites for Automatic Comment Generation. In *Proc. ASE*, pages 562–567, 2013.

[33] A. T. T. Ying and M. P. Robillard. Code Fragment Summarization. In *Proc. ESEC/FSE*, 2013, pages 655–658, 2013.