Mining API Usage Scenarios from Stack Overflow

Gias Uddin, Foutse Khomh, and Chanchal K Roy

McGill University, Polytechnique Montreal, and Unversity of Saskatchewan, Canada.

Abstract

Context: APIs play a central role in software development. The seminal research of Carroll et al. [11] on minimal manual and subsequent studies by Shull et al. [57] showed that developers prefer task-based API documentation instead of traditional hierarchical official documentation (e.g., Javadoc). The Q&A format in Stack Overflow offers developers an interface to ask and answer questions related to their development tasks.

Objective: With a view to produce API documentation, we study automated techniques to mine API usage scenarios from Stack Overflow.

Method: We propose a framework to mine API usage scenarios from Stack Overflow. Each task consists of a code example, the task description, and the reactions of developers towards the code example. First, we present an algorithm to automatically link a code example in a forum post to an API mentioned in the textual contents of the forum post. Second, we generate a natural language description of the task by summarizing the discussions around the code example. Third, we automatically associate developers reactions (i.e., positive and negative opinions) towards the code example to offer information about code quality.

Results: We evaluate the algorithms using three benchmarks. We compared the algorithms against eight state of the art techniques. Our algorithms outperformed each baseline. We developed an online tool by automatically mining API usage scenarios from Stack Overflow. A user study of 31 software developers shows that the participants preferred the mined usage scenarios in Opiner over API official documentation. The tool is available online at: http://opiner.polymtl.ca/.

Conclusion: With a view to produce API documentation, we propose a framework to automatically mine API usage scenarios from Stack Overflow, supported by three novel algorithms. We evaluated the algorithms against a total of eight state of the art baselines. We implement and deploy the framework in our proof-of-concept online tool, Opiner.

Keywords: API, Mining, Usage, Documentation.

1. Introduction

In 1987, the seminal research of Carroll et al. [11] introduced 'minimal manual' by advocating the redesigning of traditional documentation around tasks, i.e., describe the software components within the contexts of development tasks. They observed that developers are more productive while using those manuals. Since then this format is proven to work better than the traditional API documentation [5, 56, 36]. APIs (Application Programming Interfaces) offer interfaces to reusable software components. In 2000, Shull et al. [57] compared traditional hierarchical API documentation (e.g., Javadocs) against example-based documentation, each example corresponding to a development task. They observed that the participants quickly moved to task-based documentation to complete their development tasks. However, task-based documentation format is still not adopted in API official documentation (e.g., Javadocs).

Indeed, despite developers' reliance on API official documentation as a major resource for learning and using APIs [52], the documentation can often be incomplete, incorrect, and not usable [71]. This observation leads to the question of how we can improve API documentation if the only people who can accomplish this task are unavailable to do it. One potential way is to produce API documentation by leveraging the crowd [62], such as mining API usage scenarios from online Q&A forums where developers discuss how they can complete development tasks using APIs. Although these kinds of solutions do not have the benefit of authoritativeness, recent research shows that developers leverage the reviews about APIs to determine how and whether an API can be selected and used, as well as whether a provided code example is good enough for the task for which it was given [68, 67, 34]. Thus, the combination of API reviews and code examples posted in the forum posts may constitute an acceptable expedient in cases of rapid evolution or depleted development resources, offering ingredients to on-demand task-centric API documentation [55].

In this paper, with a view to assist in the automatic generation of task-based API documentation, we propose to automatically mine code examples

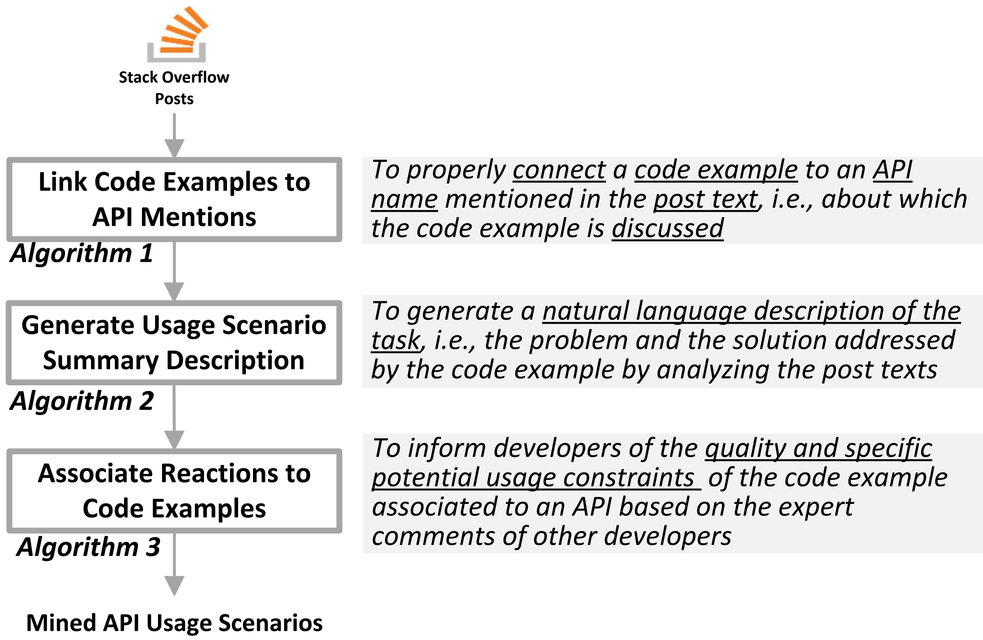

Figure 1: Our API usage scenario mining framework from Stack Overflow with three proposed algorithms

associated to different APIs and their relevant task-based usage discussions from Stack Overflow. We propose an automated mining framework that can be leveraged to automatically mine API usage scenarios from Stack Overflow. To effectively mine API usage scenarios from Stack Overflow with high performance, we have designed and developed three algorithms within our proposed framework. In Figure 1, we offer an overview of the three algorithms and show how they are used in sequence to automatically mine API usage scenarios from Stack Overflow.

• **Algorithm 1. Associate Code Examples to API Mentions.** A code snippet is provided in a forum post to complete a development task. Given a code snippet found in a forum post, we first need to link the snippet to an API about which the snippet is provided. Consider the two snippets presented in Figure 2. Both of the snippets use multiple types and methods from the java.util API. In addition, the first snippet uses the java.lang API. However, both snippets are related to the conversion of JSON data to JSON object. As such, the two snippets introduce two open source Java APIs to complete the task (Google GSON in snippet 1 and org.json in snippet 2). The state of art traceability techniques to link code examples in forum posts [62, 14, 51] will link the scenarios to both the utility (i.e., java.util, java.lang) and the open source APIs. For example, the techniques will link the first scenario to all the three APIs (java.util, java.lang, and GSON APIs), even though the scenario is actually provided to discuss the usage of GSON API. This focus is easier to understand when we look at the textual contents that describe the usage scenario.

Our algorithm links a code example to an API mentioned in the textual contents of forum post. For example, we link the fist snippet in Figure 2 to the API GSON and the second to the API org.json. We do this by observing that both GSON and org.json are mentioned in the textual contents of the post, as well as the code examples consist of class and methods from the two APIs, respectively. We adopt the definition of an API as originally proposed by Martin Fowler, i.e., a "set of rules and specifications that a software program can follow to access and make use of the services and resources provided by its one or more modules" [76]. This definition allows us to consider a Java package as an API. For example, in Figure 2, we consider the followings as APIs: 1. Google GSON, 2. Jackson, 3. org.json, 4. java.util, and 5. java.lang. Each API package thus can contain a number of modules and elements (e.g., class, methods, etc.). This abstraction is also consistent with the Java official documentation. For example, the `java.time` packages are denoted as the Java date APIs in the new JavaSE official tutorial [42]). As we observe in Figure 2, this is also how APIs can be mentioned in online forum posts.

• **Algorithm 2. Generate Textual Task Description.** Given that each code snippet is provided to complete a development task, a textual description of the task as provided in forum posts is necessary to learn about the task as well as the underlying contexts (e.g., specific API version). To offer a task-based documentation for a given code snippet that is linked to an API, we made two design decisions: 1. **Title.** We associate each code example with the title of the question, e.g., the title of a thread in Stack Overflow. 2. **Description.** We associate relevant texts from both answer (where the code example is found) and question posts. For example, in Figure 2, the first sentence ("check website ...") is not important to learn about the tasks (i.e., JSON parsing). However, for the first snippet, all the other sentences before snippet 1 are necessary to learn about the solution (because they are all related to the API GSON that is linked to snippet 1). In addition, the problem description as addressed by the task can be found in the question

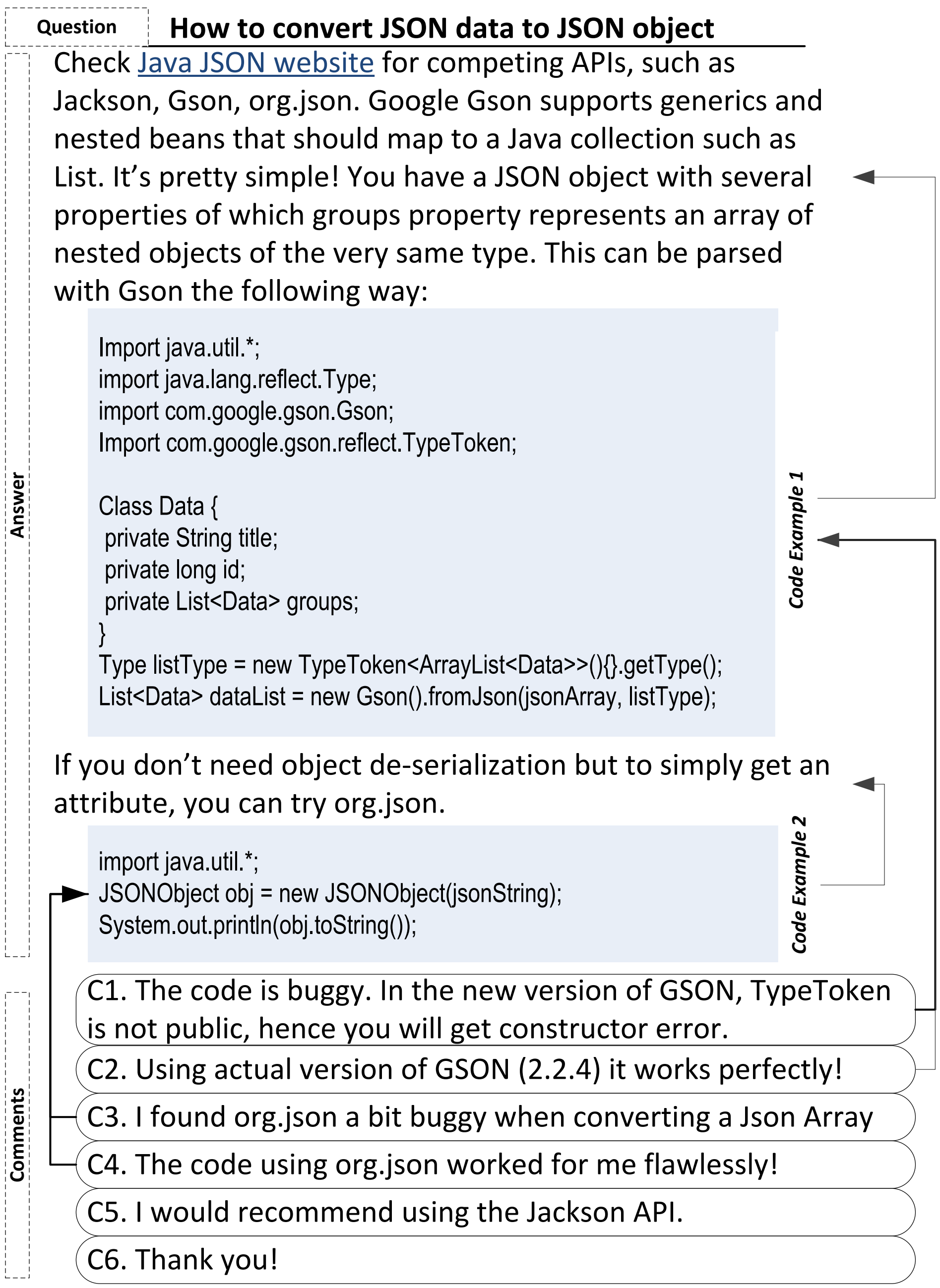

Figure 2: How API usage scenarios are discussed in Stack Overflow.

title and post. Therefore, our algorithm takes as input all the texts from answer and question posts and outputs a summary of those textual contents based on an adaptation of the popular TextRank [38] algorithm. As explained in Section 2, the TextRank algorithm is based on an adaptation of Google PageRank algorithm, which creates a graph of nodes and edges in a graph and ranks the nodes in the graph based on their association with other nodes. In our algorithm, we first heuristically find sentences relevant to an API in the textual contents. We then further refine their relevance by creating a graph of the sentences where each sentence is a node. We compute association between sentences in the graph using cosine similarity. This two-stage sentence selection process based on TextRank is useful to identify sentences relevant to the API task description. Indeed, TextRank is proven to generate high quality and relevant textual summary [38].

• **Algorithm 3. Associate Reactions to a Code Example.** As noted before reviews about APIs can be useful to learn about specific nuances and usage of the provided code examples [68, 67]. Consider the reactions in the comments in Figure 2. Out of the six comments, two (C1, C2) are associated with the first scenario and two others (C3, C4) with the second scenario. The first comment (C1) complains that the provided scenario is not buggy in the newer version of the GSON API. The second comment (C2) confirms that the usage scenario is only valid for GSON version 2.2.4. The third comment (C3) complains that the conversion of JsonArray using org.json API is a bit buggy, but the next comment (C4) confirms that scenario 2 (i.e., the one related to org.json API) works flawlessly. Given a code example, our proposed algorithm associates relevant reactions based on heuristics, such as mentions of the linked API in a reaction (e.g., In Figure 2, C1 mentions the API GSON, which is linked to code snippet 1).

We evaluated the algorithms using three benchmarks that we created based on inputs from a total of six different human coders. The first benchmark consists of 730 code examples from Stack Overflow forum posts, each manually associated with an API mentioned in the post where the code example was found. We use the first benchmark to evaluate our Algorithm 1, i.e., associate code examples to API mentions. A total of three coders participated in the benchmark creation process. We use the second benchmark to evaluate our proposed Algorithm 2, i.e., generate textual task description addressed by a code example in Stack Overflow. The second benchmark consists of 216 code examples out of the 730 code examples that we used for the

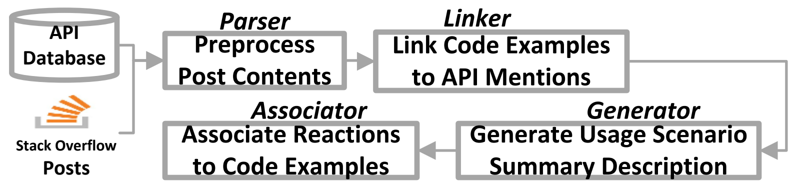

Figure 3: The major components of our API usage scenario mining framework

first benchmark. The 216 code examples were found in answer posts in Stack Overflow. The natural language summary of each of the 216 code examples was manually created based on consultations from two human coders. We use the third benchmark to evaluate our Algorithm 3, i.e, associate positive and negative reactions to a code example. The third algorithm was created by manually associated all the reactions to each of the 216 code examples that we use for the second benchmark. A total of three human coders participated in the benchmark creation process. The first author was the first coder in all the three benchmarks.

We observed precisions of 0.96, 0.96, and 0.89 and recalls of 1.0, 0.98, and 0.94 with the linking of a code example to an API mention, the produced summaries, and the association of reactions to the code examples. We compared the algorithms against eight state of the art baselines. Our algorithms outperformed all the baselines. We deployed the algorithms in our online tool to mine task-based documentation from Stack Overflow. We evaluated the effectiveness of the tool by conducting a user study of 31 developers, each completed four coding tasks using our tool, API official documentation, Stack Overflow, and search engine. The developers wrote more correct code in less time and less effort using our tool.

2. The Mining Framework

We designed our framework to mine task-based API documentation by analyzing Stack Overflow, a popular forum to discuss API usage. The framework takes as input a forum post and outputs the usage scenarios found in the post. For example, given as input the forum post in Figure 2, the framework returns two task-based API usage scenarios: (1) The code example 1 by associating it to the API Google GSON, the two comments (C1, C2) as reactions, and a description of the code example in natural language to inform of the specific development task addressed by the code example. (2) The code example 2 by associating it to the API org.json, the two comments (C3, C4) as reactions, and a summary description.

Our framework consists of five major components (Figure 3):

1. An **API database** to identify the API mentions.
2. A suite of **Parsers** to preprocess the forum post contents.
3. A **Linker** to associate a code example to an API mention.
4. A **Generator** to produce a textual task description.
5. An **Associator** to find reactions towards code examples.

2.1. API Database

An API database is required to infer the association between a code example and an API mentioned in forum post text. Our database consists of open source and official Java APIs. An open-source API is identified by a name. An API consists of one or more modules. Each module can have one or more packages. Each package contains code elements (class, method). As noted in Section 1, we consider an official Java package as an API. For each API, we record the following meta-information: (1) the name of the API, (2) the dependency of the API on other APIs, (3) the names of the modules of the API, (4) the package names under each module, (5) the type names under each package, and (6) the method names under each type. The last three items (package, type, and method names) can be collected from either the binary file of an API (e.g., a jar) or the Javadoc of the API. We obtained the first three items from the pom.xml files of the open-source APIs hosted in online Maven Central repository. Maven Central is the primary source for hosting and searching for Java APIs with over 70 million downloads every week [18].

2.2. Preprocessing of Forum Posts

Given as input a forum post, we preprocess its content as follows: (1) We categorize the post content into two types: (a) *code snippets*; [1] and (b) sen-

[1]We detect code snippets as the tokens wrapped with the <code> tag.

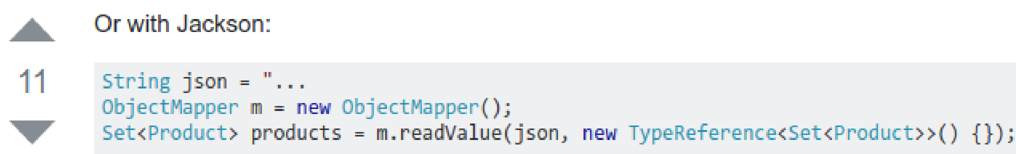

Figure 4: A popular scenario with a syntax error (Line 1) [43]

tences in the *natural language text*. (2) Following Dagenais and Robillard [14], we discard the following *invalid* code examples based on Language-specific naming conventions: (a) Non-code snippets (e.g., XML), (b) Non-Java snippets (e.g., JavaScript). We consider the rest of the code examples as *valid*.

• **Hybrid Code Parser.** We parse each valid code snippet using a hybrid parser combining ANTLR [50] and Island Parser [39]. We observed that code examples in the forum posts can contain syntax errors which an ANTLR parser is not designed to parse. However, such errors can be minor and the code example can still be useful. Consider the code example in Figure 4. An ANTLR Java parser fails at line 1 and stops there. However, the post was still considered as helpful by others (upvoted 11 times). Our hybrid parser works as follows: 1. We split the code example into individual lines. For this paper, we focused only on Java code examples. Therefore, we use semi-colon as the line separator indicator. 2. We parse each line using the ANTLR parser by feeding it the Java grammar provided by the ANTLR package. If the ANTLR parser throws an exception citing parsing error, we use our Island Parser.

• **Parsing Code Examples.** We identify API elements (types and methods) in a code example in three steps.

1. Detect API Elements: We detect API elements using Java naming conventions, similar to previous approaches (e.g., camel case for Class names) [14, 53]. We collect types that are not declared by the user. Consider the first code example in Figure 2. We add **`Type`**, **`Gson`** and **`TypeToken`**, but not **`Data`**, because it was declared in the same post: **`class Data`**.

2. Infer Code Types From Variables: An object instance of a code type declared in another post can be used without any explicit mention of the code type. For example, consider the example: `Wrapper = mapper.readValue(jsonStr, Wrapper.class)`. We associate the `mapper` object to the `ObjectMapper` type, because it was defined in another post of the same thread as: `ObjectMapper mapper = new ObjectMapper()`.

3. Generate Fully Qualified Names (FQNs): For each valid

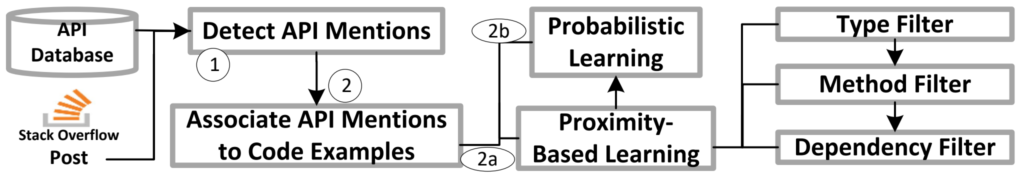

Figure 5: The components to link a scenario to API mention

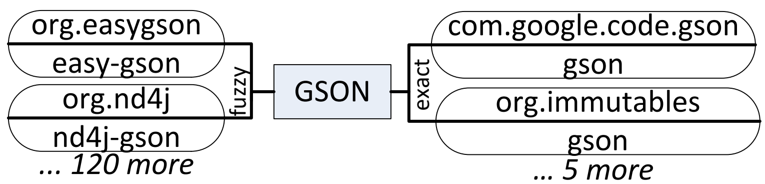

Figure 6: Partial Mention Candidate List of GSON in Figure 2

type detected in the parsing, we attempt to get its fully qualified name by associating it to an import type in the same code example. Consider the following example:

```
import com.restfb.json.JsonObject;
JsonObject json = new JsonObject(jsonString);
```

We associate `JsonObject` to `com.restfb.json.JsonObject`. We leverage both the fully and the partially qualified names in our algorithm to associate code examples to API mentions.

2.3. Associating Code Examples to API Mentions

Given as input a code example in a forum post, we associate it to an API mentioned in the post in two steps (Figure 5):

Step 1. Detect API Mentions

We detect API mentions in the textual contents of forum posts following Uddin and Robillard [72]. Therefore, each API mention in our case is a token (or a series of tokens) if it matches at least one API or module name. Similar to [72], we apply both exact and fuzzy matching. For example, for API mention ‘Gson’ in Figure 2, an exact match would be the ‘gson’ module in the API ‘com.google.code.gson’ and a fuzzy match would be the ‘org.easygson’ API. For each such API mention, we produce a Mention Candidate List (MCL), by creating a list of all exact and fuzzy matches. For example, in Figure 6, we show a partial Mention Candidate List for the mention ‘gson’. Each rectangle denotes an API candidate with its name at the top and one or more module names at the bottom (if module names matched).

For each code example, we create three buckets of API mentions: **(1)** *Same Post Before* B_b*:* each mention found in the same post, but before the code snippet. **(2)** *Same post After* B_a: each mention found in the same post, but after the code snippet. **(3)** *Same thread* B_t: all the mentions found in the title and in the question. Each mention is accompanied by a Mention Candidate List, i.e., a list of APIs from our database.

Step 2. Associate Code Examples to API Mentions

We associate a code example in a forum post to an API mention by learning how API elements in the code example may be connected to a candidate API in the mention candidate lists of the API mentions. We call this *proximity-based* learning, because we start to match with the API mentions that are more closer to the code example in the forum before considering the API mentions that are further away. For well-known APIs, we observed that developers sometimes do not mention any API name in the forum texts. In such cases, we apply *probabilistic learning*, by assigning the code snippet to an API that could most likely be discussed in the snippet based on the observations in other posts.

• **Proximity-Based Learning** uses Algorithm 1 to associate a code example to an API mention. The algorithm takes as input two items: 1. The code example C, and 2. The API mentions in the three buckets: before the code example in the post B_b, after the code example in the post B_a, and in the question post of the same thread B_t. The output from the algorithm is an association decision as a tuple $(d_{mention}, d_{api})$, where $d_{mention}$ is the API mention as found in the forum text (e.g., GSON for the first code example in Figure 2) and d_{api} is the name of the API in the mention candidate list of the API mention that is used in the code example (e.g., `com.google.code.gson` for the first code example in Figure 2).

The algorithm uses three filters (L1, discussed below). Each filter takes

input : (1) Code Example $C = (T, E)$, (2) API Mentions in buckets $B = (B_b, B_a, B_t)$
output: Association decision, $D = \{d_{mention}, d_{api}\}$

Proximity Filters $F = [F_{type}, F_{method}, F_{dep}]$;
$D = \emptyset$, $N =$ `length(`B`)`, $K =$ `length(`F`)`;
for $i \leftarrow 1$ *to* N **do**
    $B_i = B[i]$, $H =$ `getMentionApiTuples` (B_i);
    for $k \leftarrow 1$ *to* K **do**
        Filter $F_k = F[k]$, $H =$ `getHits` (F_k, C, H, L_i);
        if $|H| = 1$ **then** $D = H[1]$; **break**;
procedure `getMentionApiTuples(`B`)`
    List$<$ `MentionAPI` $>$ $M = \emptyset$;
    foreach *Mention* $m \in B$ **do**
        $MCL = \{a_1, a_2, \dots a_n\}$ ▷ MCL of m;
        foreach *API* $a_i \in MCL$ **do**
            `MentionAPI` ma $= \{m, a_i\}$; M.`add` (ma)
    return M;
procedure `getHits` (F_k, C, H)
    $S = \emptyset$;
    for $i \leftarrow 1$ *to* `length` (H) **do**
        $S[i]$ = compute score of $H[i]$ for C using F_k;
    if `max` $(S) = 0$ **then** **return** H;
    else
        $H_{new} = \emptyset$;
        for $i \leftarrow 1$ *to* `length` (H) **do**
            if $S[i] =$ `max` (S) **then** H_{new}.`add` $(H[i])$;
        return H_{new}
return D

Algorithm 1: Associate a code example to an API mention as input a list of tuples in the form (mention, candidate API). The output from the filter is a set of tuples, where each tuple in the set is ranked the highest based on the filter. The higher the ranking of a tuple, the more likely it is associated to the code example based on the filter. For each mention bucket (starting with B_b, then B_a, followed by B_t), we first create a list of tuples H using `getMentionApiTuples` (L4, L8-14). Each tuple is a pair of API mention and a candidate API. We apply the three filters on this list of tuples. Each filter produces a list of hits (L6) using `getHits` procedure (L15-24). The output from a filter is passed as an input to the next filter, following the principle of *deductive learning* [62]. If the list of hits has only one tuple, the algorithm stops and the tuple is returned as an association decision (L7).

F1. Type Filter. For each code type (e.g., a class) in the code example, we search for its occurrence in the candidate APIs from Mention Candidate List. We compute type similarity between a snippet s_i and a candidate c_i as follows.

$$\text{Type Similarity} = \frac{|\text{Types}(s_i) \bigcap \text{Types}(c_i)|}{|\text{Types}(s_i)|} \tag{1}$$

Types(s_i) is the list of types for s_i in bucket. Types(c_i) is the list of the types in Types(s_i) that were found in the types of the API. We associate the snippet to the API with the maximum type similarity. In case of more than one such API, we create a *hit list* by putting all those APIs in the list. Each entry is considered as a potential hit.

F2. Method Filter. For each of candidate APIs returned in the list of hits from type filter, we compute method similarity between a snippet s_i and a candidate c_i:

$$\text{Method Similarity} = \frac{|\text{Methods}(s_i) \bigcap \text{Methods}(c_i)|}{|\text{Methods}(s_i)|} \tag{2}$$

We associate the snippet to the API with the maximum similarity. In case of more than one such API, we create a *hit list* of all such APIs and pass it to the next filter.

F3. Dependency Filter. We create a *dependency graph* by consulting the dependencies of APIs in the hit list. Each node corresponds to an API from the hit list. An edge is established, if one API depends on another API. From this graph, we find the API with the maximum number of incoming

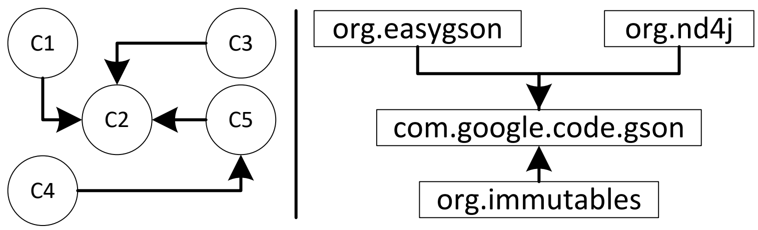

Figure 7: Dependency graph given a hit list

edges, i.e., the API on which most of the other APIs depend on. If there is just one such API, we assign the snippet to the API. This filter is developed based on the observation that developers mention a popular API (e.g., one on which most other APIs depend on) more frequently in the forum post than its dependents.

In Figure 7, we show an example dependency graph (left) and a partial dependency graph for the four candidate APIs from Figure 6 (right). In the left, both C2 and C5 have incoming edges, but C2 has maximum number of incoming edges. In addition, C5 depends on C2. Therefore, C2 is most likely the *core* and most popular API among the five APIs. The dependency filter is useful when a code example is short, with generic type and method names. In such cases, the code example can potentially match with many APIs. Consider a shortened version of the first code example in Figure 2:

```
import com.google.code.Gson;
Data json = new Gson().fromJson(string, Data.class);
```

Both the type (com.google.code.Gson) and methods (Gson() and fromJson(. . .)) can be found in the two APIs in Figure 6: org.immutables and com.google.code.gson. However, as we see in Figure 7 (right), all the APIs depend on com.google.code.gson. Therefore, we assign the snippet to the mention Gson and the API com.google.code.gson.

• **Probabilistic Learning** is used when an API mention is not found in post texts, i.e., we cannot link a code example to an API using proximity learning. In such cases, we associate a code example to an API that was most frequently associated in other code examples. We do this by computing the *coverage* of an API across those code examples linked by the proximity learning. A coverage is the total number of times the types of an API is found in those snippets. Suppose, for four code examples C1-C4, C1 and C2

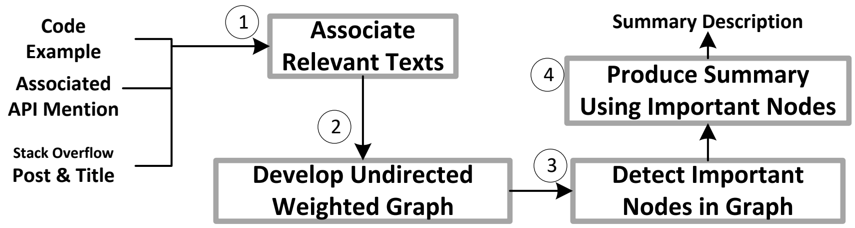

Figure 8: Steps to produce summary description of a scenario

are already linked to API A1, and C3 to API A2, but no API is mentioned in the post where C4 is found. In such cases, we compute the coverage of types in C4 (say T1, T2) in the linked snippets. If T1 is present in C1 and C2, and T2 in C3, we have coverage of 2 for API A1, and coverage of 1 for API A2. Thus, we link C4 to API A1. This learning is based on two observations: (1) developers tend to refer to the same API types in many different forum posts, and (2) when an API type is well-known, developers tend to refer to it in the code examples without mentioning the API (see for example [44]).

2.4. Generating Natural Language Task Description

We produce textual description for code examples that are found in the answer posts, because such a code example is in need to be understood for a development task [62]. Our algorithm is based on the TextRank algorithm [38]. Our algorithm operates in four steps (Figure 8):

1. **Associate Relevant Texts.** We produce an input as a list of sentences from the forum post where the code example is found. Each sentence is selected by considering its proximity from the API mention linked to the code example. For example, for the first code example in Figure 2 linked to the API Gson, we pick all the sentences before the code example except the first one. To pick the sentences, we apply beam-search. We start with the first sentence in the forum post where API is mentioned. We then pick next possible sentence by looking for two types of signals: (a) it refers to the API (e.g., using a pronoun), and (b) it refers to an API feature. To identify features, we use noun phrases based on shallow parsing [31]. By adhering to the principle of task-oriented documentation, we organize the relevant texts into three parts: (a) **Task Title**. The one line description of the task, as

found in the title of the question. (b) **Problem.** The relevant texts obtained from the question that describe the specific problem related to the task. (c) **Solution.** The relevant texts obtained from the answer where the code example is found. We produce a summarized description by applying Steps 2 and 3 once for 'Problem' texts and another for the 'Solution' texts.

2. **Develop Undirected Weighted Text Graph.** We remove stop words from each input sentence and then vectorize the sentence into textual units (e.g., ngram). We compute the distance between two sentences. A distance is defined as (1 - similarity). Similarity can be detected using standard metrics, such as cosine similarity. An edge is established between two sentences, if they show some similarity between them. The weight of each edge is the computed distance.

3. **Detect Important Nodes in Graph.** We traverse the text graph using the PageRank algorithm to find optimal weight for each node in the graph by repeatedly iterating over the following equation (until no further optimization is possible):

$$WS(V_i) = (1 - d) * \sum_{V_j \in (V_i)} \frac{w_{ji}}{\sum_{v_k \in Out(V_j)} w_{jk}} WS(V_j) \tag{3}$$

Here d is the damping factor, V are nodes, WS are the weights. $\in (V_i)$ are the incoming edges to node V_i.

4. **Produce Summary Using Important Nodes.** In order to produce the summary using important nodes, we first pick the top N nodes with the most weights among all the nodes. We then rank the nodes based on their appearance in the original post (i.e., problem or solution). Each node essentially corresponds to a sentence in the post. We then combine all the ranked sentences to produce the summary.

Finally, we produce a description by combining the three items in order, i.e., Title, Problem and Solution summaries.

2.5. Associating Reactions to Usage Scenarios

The final part of our proposed framework is to associate reactions to the usage scenarios. In order to do this, we first gather all the comments of the post where the code example is found. We then use the principles of discourage learning [35] to associate the reactions in the comments (i.e., negative and positive opinions) towards the code examples. The inputs to the algorithm are all the comments towards the post where the code example is found. Our algorithm works as follows. 1. We sort the comments in the time of posting. The earliest comment is placed at the top. We identify opinionated sentences in each comment. 2. We identify the API mentions in each comment. 3. We label an opinionated comment as relevant to an API mention if it refers to the API mention by name or by pronoun. To determine whether a pronoun refers to an API mention, we determine the distance between the API mention and the pronoun and whether another API was mentioned in between. If the opinionated comment is related to the API mention associated to the code example, we associate the comment to the code example. For example, in Figure 2, the comment C4 is not considered as relevant to the code example 1, because the closest and most recent API name to the comment is the org.json API in comment C3. 4. For opinionated comments that do not directly/indirectly refer to an API mention (e.g., using pronoun), we associate those to the code example based on a notion called *implicit reference*. We consider a comment as implicitly related to the code example, if no other APIs are mentioned at least two comments above it. To analyze the opinionated sentences, our algorithm can use the output of any sentiment detection tools. The current framework uses an adaptation of the Domain Sentiment Orientation algorithm as originally proposed by Hu et al. [25]. The algorithm was previously adopted by Google to analyze local service reviews [4]. The algorithm showed more precision than other sentiment detection tools to detect the opinionated sentences in Stack Overflow [70].

3. Evaluation

We extensively evaluated the feasibility of our mining framework by investigating the accuracy of the three proposed algorithms. In particular, we answer the following three research questions:

1. What is the performance of the algorithm to link code examples to APIs mentioned in forum texts?

2. What is the performance of generating the natural language summary for a scenario?

3. What is the performance of linking the reactions (the positive and negative opinions) to a scenario?

Both high precision and recall are required in the mining of scenarios. A precision in the linking of a scenario to an API mention ensures we do not link a code example to a *wrong* API, a high recall ensures that we do not miss many usage scenarios relevant to an API. Similarly, a high precision and a high recall are required to associate reactions to a code example. For the summary description of a code example, a high precision is more important because otherwise we associate a wrong description to a code example.

Given that all our three proposed algorithms are information retrieval in nature, we report four standard evaluation metrics (Precision P, Recall R, F1-score $F1$, and Accuracy A) as follows:

$$P = \frac{TP}{TP + FP},\ R = \frac{TP}{TP + FN},\ F1 = 2 * \frac{P * R}{P + R}, A = \frac{TP + TN}{TP + FP + TN + FN}$$

$TP =$ Nb. of true positives, and $FN =$ Nb. false negatives.

Evaluation Corpus. We analyze the Stack Overflow threads tagged as 'Java+JSON', i.e., the threads contained discussions related to the JSON-based software development tasks using Java APIs. We selected the Java JSON-based APIs because JSON-based techniques support diverse development scenarios, such as, both specialized (e.g., serialization) as well as utility-based (e.g., lightweight communication), etc. We used the 'Java+JSON' threads from Stack Overflow dump of 2014 for the following reasons:

1. It offers a rich set of competing APIs with diverse usage discussions, as reported by other authors previously [68].

2. It allowed us to also check whether the API official documentation were updated with scenarios from the dataset (see Section 4). Intuitively, our mining framework is more useful when the framework can be used to update API official documentation by automatically mining the API usage scenarios, such as when the official documentation is found to be not updated with the API usage scenarios discussed in Stack Overflow even when sufficient time is spent between when such as scenario is discussed in Stack Overflow and when an API official documentation is last updated.

In Table 1 we show descriptive statistics of the dataset. There were 22,733 posts from 3,048 threads with scores greater than zero. Even though questions were introduced during or before 2014, each question is still active in

Table 1: Descriptive statistics of the dataset (Valid Snippets)

Threads	Posts	Sentences	Words	Snippet	Lines	Users
3048	22.7K	87K	1.08M	8596	68.2K	7.5K
Average	7.5	28.6	353.3	2.8	7.9	3.9

Stack Overflow, i.e., the underlying tasks addressed by the questions are still relevant. There were 8,596 *valid* code snippets and 4,826 invalid code snippets. On average each valid snippet contained at least 7.9 lines. The last column "Users" show the total number of distinct users that posted at least one answer/comment/question in those threads.

We evaluated our proposed three algorithms by creating three benchmarks out of our evaluation corpus. In our previous research of two surveys of 178 software developers, we found that developers consider the combination of code examples and reviews from other developers towards the code examples in online developer forums (e.g., Stack Overflow) as a form of API documentation. We also found that developers use such documentation to support diverse development tasks (e.g., bug fixing, API selection, feature usage, etc.) [66]. Therefore, it is necessary that our mining framework is capable of supporting any development scenario. This can be done by linking any code example to an API mention, and by producing a task-based documentation of an API to support any development task. Therefore, to create the benchmarks from the evaluation corpus, we pick code examples using random sampling that offers representation of the diverse development scenarios in online developer forums in general without focusing on a specific development scenario (e.g., How-to, bug-fixing) [27, 80].

3.1. RQ_1 Performance of Linking Code Example to API Mention

3.1.1. Approach

We assess the performance of our algorithm to link code examples to API mentions using a benchmark that consists of randomly selected 730 code examples from our entire corpus. 375 code examples were sampled from the 8589 valid code snippets and 355 from the 4826 code examples that were labeled as invalid by the *invalid code detection* component of our framework. The size of each subset (i.e., valid and invalid samples) is determined to capture a statistically significant snapshot of our entire dataset (95% confidence interval). The evaluation corpus was manually validated by three coders:

Table 2: Analysis of agreement among the coders to validate the association of APIs to code examples (Using Recal3 [20])

	Kappa (Pairwise)	Fleiss	Percent	Krippen α
Overall	0.97	0.97	99.4%	0.97
Valid	0.93	0.93	98.7%	0.93
Discarded	1.0	1.0	100%	1.0

The first two coders are the first two authors of this paper. The third coder is a graduate student and is not a co-author. The benchmark creation process involved three steps: (1) The three coders independently judged randomly selected 80 code examples out of the 730 code examples: 50 from the valid code examples and 30 from the invalid code examples. (2) The agreement among the coders was calculated, which was near perfect (Table 2): pairwise Cohen κ was 0.97 and the percent agreement was 99.4%. To resolve disagreements on a given code example, we took the majority vote. (3) Since the agreement level was near perfect, we considered that any of the coders could complete the rest of the coding without introducing any subjective bias. The first coder then labeled the rest of the code examples. The manual assessment found nine code examples as invalid. We labeled our algorithm as wrong for those, i.e., false positives. In the end, the benchmark consisted of 367 valid and 363 invalid code examples.

• **Baselines.** We compare our algorithm against two baselines: (B1) Baker [62], and (B2) Google search. We describe the baselines below.

B1. Baker: As noted in Section 1, related techniques [62, 51, 14] find fully qualified names of the API elements in the code examples. Therefore, if a code example contains code elements from multiple APIs, the techniques link the code example to all APIs. We compare our algorithm against Baker, because it is the state of the art technique to leverage an API database in the linking process (unlike API usage patterns [51]). Given that Baker was not specifically designed to address the type of problem we attempt to address in this paper, we analyze both the originally proposed algorithm of Baker as well as an enhanced version of the algorithm to ensure fair comparison.

Baker (Original). We apply the original version of the Baker algorithm [62] on our benchmark dataset as follows.

1. Code example consisting of code elements (type, method) only from one API: We attempt to link it using the technique proposed in Baker [62].

2. Code example consisting of code elements from more than one API: if the code example is associated to one of the API mentioned in the post, we leave it as 'undecided' by Baker.

Baker (Major API). For the 'undecided' API mentions by Baker (Original), we further attempt to link an API as follows. For a code example where Baker (original) could not decide to link it to an API mention, we link it to an API that was used the most frequently in the code example. We do this by computing the call frequency of each API in the code example. Suppose, we model a code example as an API call matrix $A \times T$, where A stands for an API and T stands for a type (class, method) of the API that is reused in the code example. The cell (A_i, T_j) has a value 1 if type T_j from API A_i is called in the code example. We compute the reuse frequent of each API A_i using the matrix by summing the number of distinct calls (to different types) is made in the code example. Thus $S_i = \sum_{j=1}^{m} T_j$. We assign the code example to the API A_i with the maximum S_i among all APIs reused.

B2. Search: In our previous study of two surveys involving 178 software developers [67], we found that developers frequently use Google search engine to find for solutions in Stack Overflow. This finding is not surprising, because the use of search engines to find solutions for development tasks as well API coverage is well-documented in software engineering literature [49, 47]. Indeed, search engines are extensively used in development activities involving API usage [16]. Therefore, we search each valid code example in Google. We check the first three hits (without advertisement) per result. If at least one hit contains a reference to the associated API, we label the result as correct. We do not consider a result relevant if it points to the same Stack Overflow post where the code example is found.

In the Google search engine, we do not preprocess a code example. Our decision to use a code example as is was motivated by previous findings that developers use Google to support diverse development tasks in their everyday tasks and that they do not have access to a specialized engine to meet their diverse search needs [67]. Therefore, in the absence of any other tool that could be available to software developers to refine/preprocess a code example before using Google for code search, it is safe to assume that developers would most likely use a code example as it is. Therefore, this setting could give us the most unbiased insight of the current state of code to API linking search in daily development activities.

Table 3: Performance of linking code examples to API Mentions

Proposed Algorithm	Precision	Recall	F1 Score	Acc
Detect Invalid	-	-	-	0.97
Link Valid w/Partial info	0.94	1.0	0.97	0.94
Link Valid w/Full info	0.96	1.0	0.98	0.96
Overall w/Partial Info	0.94	0.97	0.95	0.95
Overall w/Full Info	0.96	1.0	0.98	0.96
Baselines (applied to valid code examples)				
B1a. Baker (Original)	0.97	0.49	0.65	0.48
B1b. Baker (Major API)	0.88	0.66	0.76	0.61
B2. Search (Google)	0.39	0.88	0.54	0.37

3.1.2. Results

We achieved a precision of 0.96 and a recall of 1.0 using our algorithm (Table 3). A recall of 1.0 was achieved due to the greedy approach of our algorithm which attempts to find an association for each code example. The Google search shows the lowest precision (0.39), confirming the assumption that Google is primarily a generally purpose search engine. The baseline Baker (Original) shows the best precision among all (0.97), but with the lowest recall (0.49). This level of precision corroborates with the precision reported by Baker on Android SDKs [62]. The low recall is due to the inability of Baker to link a code example to an API mention, when more than one API is used in the code example. For those code examples where Baker (Original) was undecided, we further attempted to improve Baker to find an API that is the most frequently used in the code example. The Baker (Major API) baseline improves the recall of Baker (Original) from 0.49 to 0.66. However, the precision of Baker (Major API) drops to 0.88 from 0.97. The drop in precision is due to the fact the major API is not the API for which the code example is provided. This happened due to the extensive usage of Java official APIs (e.g., java.util) in the code example, while the mentioned API in the textual content referred to an open-source API (e.g., for Jackson/org.json for JSON parsing). In some cases the major API could not be determined due to multiple API having the maximum occurrence frequency as well as the presence of *ambiguous* types in the code example. An API type is *ambiguous* in our case if more than API can have a type with the same name. For example, `JSONObject` is a popular class name among more than 900 APIs in Maven central only. Even the combination of type and method could be ambiguous. For example, the method `getValue` is common for a given type, such as `JSONObject.getValue(...)`. In such cases, the usage of API mentions in the textual contents offered our proposed algorithm an improvement in precision and recall over Baker.

We report the performance of our algorithm under different settings: 1. **Detect Invalid.** We observed an accuracy of 0.97 to detect invalid code examples. 2. **Link to valid with Partial Info.** We are able to link a valid code to an API mention with a precision of 0.94 using only the type-based filter from the proximity learning and probabilistic learning. This experimentation was conducted to demonstrate how much performance we can achieve with minimal information about the candidate APIs. Recall that the type-based filter only leverages API type names, unlike a combination of API type and method names (as used by API fully qualified name inference techniques [62, 14, 51]. Out of the two learning rules in our algorithm, Proximity learning shows better precision than Probabilistic learning (2 vs 14 wrong associations). 3. **Link to valid with Full Info.** When we used all the filters under proximity learning, the precision level was increased to 0.96 to link a valid code example to an API mention. The slight improvement in precision confirms previous findings that API types (and not methods) are the major indicators for such linking [62, 14]. 4. **Overall.** We achieved an overall precision of 0.94 and a recall of 0.97 while using partial information.

Almost one-third of the misclassified associations happened due to the code example either being written in programming languages other than Java or the code example being invalid. The following JavaScript code snippet was erroneously considered as valid. It was then assigned to a wrong API: `var jsonData; $.ajax(type: 'POST')....`

Five of the misclassifications occurred due to the code examples being very short. Short code examples lack sufficient API types to make an informed decision. Misclassifications also occurred due to the API mention detector not being able to detect all the API mentions in a forum post. For example, the following code example [45] was erroneously assigned to the `com.google.code.gson` API. However, the correct association would be the `com.google.gwt` API. The forum post (answer id 20374750) contained both API mentions. However, `com.google.gwt` was mentioned using an acronym GWT and the API mention detector missed it.

```
AutoBean<Ts> b = AutoBeanUtils.getAutoBean(ts)
return AutoBeanCodex.encode(b).getPayload();
```

3.2. RQ$_2$ Performance of Producing Textual Task Description

3.2.1. Approach

The evaluation of natural language summary description can be conducted in two ways [12]: 1. User study: participants are asked to rate the summaries 2. Benchmark: The summaries are compared against a benchmark. We follow benchmark-based settings, which compare produced summaries are compared against those in the benchmark using metrics, e.g., coverage of the sentences.

In our previous benchmark (RQ$_1$), out of the 367 valid code example, 216 code examples were found in the answer posts. The rest of the valid code examples (i.e., 151) were found in the answer posts. We assess the performance of our summarization algorithm for the 216 code examples that are found in the answer posts, because each code example is provided in an attempt to suggest a solution to a development task and our goal is to create task-based documentation support for APIs.

We create another benchmark by manually producing summary description for the 216 code examples using two types of information: 1. the description of the task that is addressed by the code example, and 2. the description of the solution as carried out by the code example. Both of these information types can be obtained from forum posts, such as problem definition from the question post and solution description from the answer post. We picked sentences following principles of extractive summarization [12] and minimal manual [11], i.e., pick only sentences that are related to the task. Consider a task description, *"I cannot convert JSON string into Java object using Gson. I have previously used Jackson for this task"*. If the provided code example is linked to the API Gson, we pick the first sentence as relevant to describe the problem, but not the second sentence. A total of two human coders were used to produce the benchmark. The first coder is the first author of this paper. The second coder is a graduate student and is not a co-author of this paper. The two coders followed the following steps: 1. create a coding guide to determine how summaries can be produced and evaluated, 2. randomly pick n code examples out of the 216 code examples, 3. produce summary description of each code example by summarizing the problem text (from question post) and the solution text (from answer post). 4. Compute the

Table 4: Agreement between the coders for RQ2 benchmark

	Iteration 1 (5)	Iteration 2 (15)	Iteration 3 (30)
Problem	60.0%	77.8%	87.1%
Solution	60.0%	87.5%	83.3%
Overall	60.0%	82.5%	85.2%

Table 5: Algorithms to produce summary description

Techniques	Precision	Recall	F1 Score	Acc
Proposed Algorithm	0.96	0.98	0.97	0.98
B1. Luhn	0.66	0.82	0.71	0.77
B2. Textrank	0.66	0.83	0.72	0.77
B3. Lexrank	0.64	0.81	0.70	0.76
B4. LSA	0.65	0.82	0.71	0.76

agreement between the coders. Resolve disagreements by consulting with each other. 5. Iterate the above steps until the coders agreed on at least 80% of the description in two consecutive iterations, i.e., after that any of the coders can produce the summary description of the rest of code examples without introducing potential individual bias. In total, the two coders iterated three times and achieved at least 82% agreement in two iterations (see Table 4). In Table 4, the number besides an iteration shows the number of code examples that were analyzed by both coders in an iteration (e.g., 30 for the third iteration). On average, each summary in the benchmark contains 5.4 sentences and 155.5 words.

• **Baselines.** We compare against four off-the-shelf extractive summarization algorithms [21]: (B1) Luhn, (B2) Lexrank, (B3) TextRank, and (B4) Latent Semantic Analysis (LSA). The first three algorithms were previously used to summarize API reviews [68]. The LSA algorithms are commonly used in information retrieval and software engineering both for text summarization and query formulation [23]. Extractive summarization techniques are the most widely used automatic summarization algorithms [21]. Our proposed algorithm utilizes the TextRank algorithm. Therefore, by applying the TextRank algorithm without the adaption that we proposed, we can estimate the impact of the proposed changes.

3.2.2. Results

We achieved the best precision (0.96) and recall (0.98) using our proposed engine that is built on top of the TextRank algorithm. Each summarization algorithm takes as input the following texts: 1. the title of the question, and 2. all the textual contents from both the question and the answer posts. By merely applying the TextRank algorithm on the input we achieved a precision 0.66 and a recall of 0.83 (i.e., without the improvement of selecting sentences using beam search that we suggested in our algorithm). The improvement in our algorithm is due to the following two reasons: 1. the selection of a smaller subset out of the input texts based on the contexts of the code example and the associated API (i.e., Step 1 in our proposed algorithm), and 2. the separate application of our algorithm on the Problem and Solution text blocks. This approach was necessary, because the baselines showed lower recall due to their selection of un-informative texts. The TextRank algorithm is the best performer among the baselines.

3.3. RQ_3 Performance of Linking Reactions to Code Examples

3.3.1. Approach

We assess the performance of our algorithm using a benchmark that is produced by manually associating reactions towards the 216 code examples that we analyzed for RQ1 and RQ2. Our focus is to evaluate the performance of the algorithm to *correctly* associate a reaction (i.e., positive and negative opinionated sentence) to a code example. As such, as we noted in Section 2.5, our framework supports the adoption of any sentiment detection tool to detect the reactions. Given that the focus of this evaluation is on the *correct* association of reactions to code examples, we need to mitigate the threats in the evaluation that could arise due to the inaccuracies in the detection of reactions by a sentiment detection tool [41]. We thus manually label the polarity (positive, negative, or neutral) of each sentence in our benchmark following standard guidelines in the literature [6, 26].

Out of the 216 code examples in our benchmark, 68 code examples from 59 answers consisted of at least one comment (total 201 comments). The 201 comments had a total of 493 sentences (190 positive, 55 negative, 248 neutral). Four coders judged the association of each reaction (i.e., positive and negative sentences) towards the code examples. For each reaction, we label it either 1 (associated to the code example) or 0 (non-associated). The association of each reaction to code example was assessed by at least two coders. The first coder (C1) is the first author, the second (C2) is a graduate

Table 6: Analysis of Agreement Between Coders To Validate the Association of Reactions to Code Examples (Using Recal2 [19])

	Total	Percent	Kappa (pairwise)	Krippen α
C1-C2	174	83.9%	0.46	0.45
C2-C3	51	62.7%	0.12	0.05
C1-C3	103	84.5%	0.50	0.51

Table 7: Performance of associating reactions to code examples

Technique	Precision	Recall	F1 Score	Acc
Proposed Algorithm	0.89	0.94	0.91	0.89
B1. All Comments	0.45	0.84	0.55	0.45
B2. All Reactions	0.74	0.84	0.78	0.74

student, third (C3) is an undergraduate student, and fourth (C4) is the second author of the paper. The second and third coders are not co-authors of this paper. The first coder coded all the reactions. The second and third coders coded 174 and 103 reactions, respectively. For each reaction, we took the majority vote (e.g., if C2 and C3 label as 1 but C1 as 0, we took 1, i.e., associated). The fourth coder (C4) was consulted when a majority was not possible. This happened for 22 reactions where two coders (C1 and C2/C3) were involved and they disagreed. The labeling was accompanied by a coding guide. Table 6 shows the agreement among the first three coders.

• **Baselines.** We compare against two baselines: (B1) All Comments. A code example is linked to all the comments. (B2) All Reactions. A code example is linked to all the positive and negative comments. The first baseline offers us insights on how well a blind association technique without sentiment detection may work. The second baseline thus includes only the subset of sentences from all sentences (i.e., B1) that are either positive or negative. However, not all the reactions may be related to a code example. Therefore, the second baseline (B2) offers us insights on whether the simple reliance on sentiment detection would suffice or whether we need a more sophisticated contextual approach like our proposed algorithm that picks a subset of the positive and negative reactions out of all reactions.

3.3.2. Results

We observed the best precision (0.89) and recall (0.94) using our proposed algorithm to link reactions to code examples. The baseline 'All Reactions' shows much better precision than the other baseline, but still lower than our algorithm. The lower precision of the 'All Reaction' is due to the presence of reactions in the comments that are not related to the code example. Such reactions can be of two types: 1. Developers offer their views of competing APIs in the comments section. Such views also generate reactions from other developers. However, to use the provided code example or complete the development task using the associated API, such discussions are not relevant. 2. Developers can also offer views about frameworks that may be using the API associated to the code example. For example, some code examples associated with Jackson API were attributed to the spring framework, because spring bundles the Jackson API in its framework. We observed that such discussions were often irrelevant, because to use the Jackson API, a developer does not need to install the Spring framework. Therefore, from the usage perspective of the snippet, such reactions are irrelevant.

4. Discussion

We implemented our framework in an online tool, Opiner [69]. Using the framework deployed in Opiner, a developer can search an API by its name to see all the mined usage scenarios of the API from Stack Overflow. We previously developed Opiner to mine positive and negative opinions about APIs from Stack Overflow. Our proposed framework in this paper extends Opiner by also allowing developers to search for API usage scenarios, i.e., code examples associated to an API and their relevant usage information.

The current version shows results from our evaluation corpus. We present the usage scenarios by grouping code examples that use the same types (e.g., class) of the API. As noted in Section 3, our evaluation corpus uses Stack Overflow 2014 dataset. This choice was not random. We wanted to see, given sufficient time, whether the usage scenarios in our corpus were included in the API official documentation. We found a total of 8596 valid code examples linked to 175 distinct APIs in our corpus. The majority of those (60%) were associated to five APIs: java.util, org.json, Gson, Jackson, java.io. Most of the mined scenarios for those APIs were absent in their official documentation, e.g., for Gson, only 25% types are used in the code examples of its official documentation, but 81.8% of the types are discussed in our mined usage scenarios. Therefore, the automatic mining of the usage scenarios using our framework can assist the API authors who could not include those in the API official documentation.

In Figure 9, we show screenshots of our tool. A user can search an API by name in ① to see the mined tasks of the API ③. An example task is shown in ④. Other relevant tasks (i.e., that use the same classes and methods of the API) are grouped under 'See Also' (⑤). Each task under the 'See Also' can be further explored (⑥). Each task is linked to the corresponding post in Stack Overflow where the code example was found (by clicking on the *details* label). The front page shows the top 10 APIs with the most mined tasks ②.

• **Effectiveness of our Tool.** Although we extensively evaluated the accuracy of our algorithms, we also measured the effectiveness of our tool with a user study. Given that the focus of evaluation of this paper is to study the accuracy of the proposed three algorithms in our mining framework and not allude on the effectiveness of Opiner as a tool, we briefly describe the user study design and results below.

Participants. We recruited 31 developers. Among them, 18 were recruited through the online professional developers site, Freelancer.com. The other participants (13) were recruited from four universities, two in Canada and two in Bangladesh. Each recruiter had professional software development experience in Java. Each freelancer was remunerated with $20.

Tasks. The developers each completed four coding tasks involving four APIs (one task for each of Jackson [17], Gson [22], Spring [58] and Xstream [73]). The four APIs were found in the list of top 10 most discussed APIs in our evaluation corpus. The four tasks were picked randomly from our evaluation corpus of 22.7K Stack Overflow posts. Each task was observed in Stack Overflow posts more than once and was asked by more than one devel- oper. Each task was related to the manipulation of JSON inputs using Java APIs for JSON parsing. For example, the task with Jackson converts a Java object to JSON format, the task with Gson converts a JSON string into a Java object, the task with Xstream converts an XML string into a JSON string, and the task with Spring converts an HTTP JSON response into a Java object.

For the user study the objects were four resources (our tool, Stack Overflow, Official documentation, Search Engines). The participants were divided into four groups. Each of first three groups (G1-3) had eight and the last

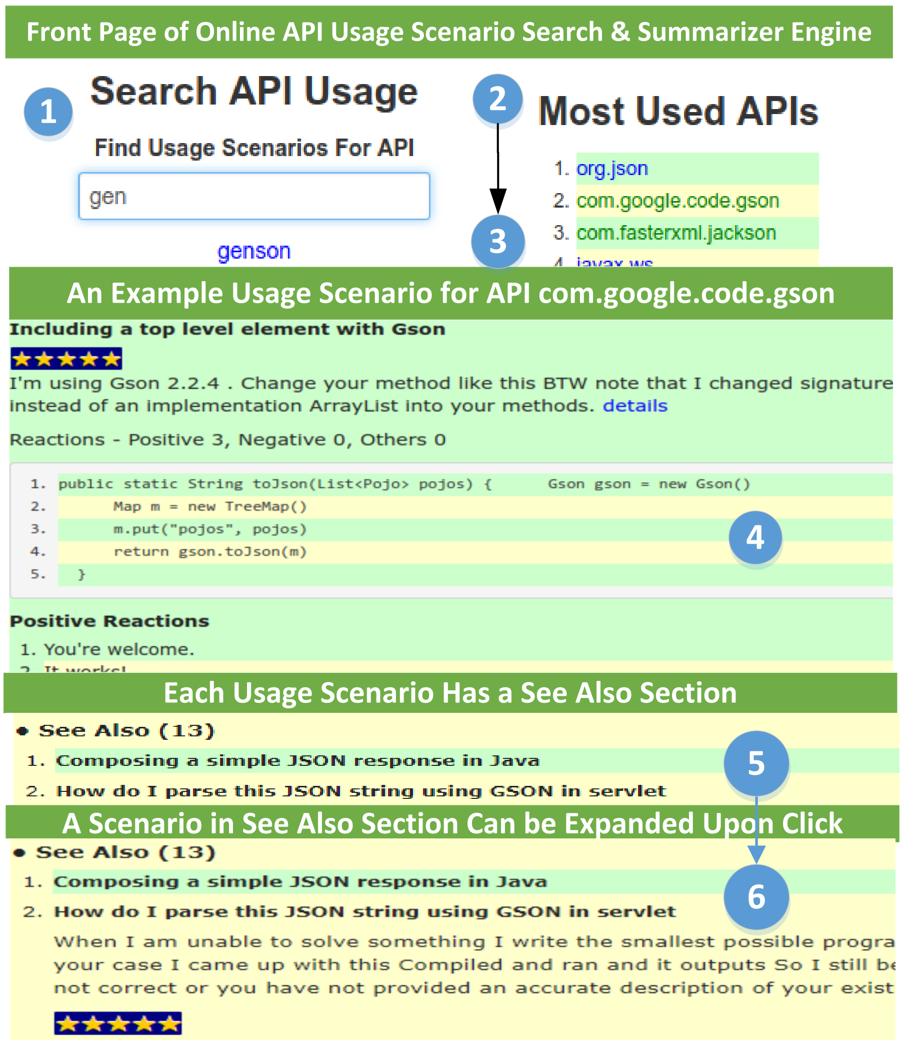

Figure 9: Screenshots of online our task-based API documentation tool

group (G4) had seven participants. Each participant in a group was asked to complete the four coding tasks. Each participant in a group completed the tasks using the four resources in the following order.

- G1: Jackson (Stack Overflow), Gson (Javadoc), Xstream (Opiner), Spring (Everything including Search Engine)
- G2: Spring (Stack Overflow), Jackson (Javadoc), Gson (Opiner), Xstream (Everything including Search Engine)
- G3: Xstream (Stack Overflow), Spring (Javadoc), Jackson (Opiner), Gson (Everything including Search Engine)
- G4: Gson (Stack Overflow), Xstream (Javadoc), Spring (Opiner), Jackson (Everything including Search Engine)

We collected the time took to complete each task and effort spent using NASA TLX index [24] (nasatlx.com). We assessed the correctness of a solution by computing the coverage of correct API elements. We summarize major findings below. More details of the study the results are provided in our online appendix [1].

While using our tool Opiner, the participants on average coded with more correctness, spent the least time and effort out of all resources. For example, using Opiner the average time developers spent to complete a coding task was 18.6 minutes and the average effort as reported in their TLX metrics was 45.8. In contrast, participants spent the highest amount of time (23.7 minutes) and effort (63.9) per coding solution when using the official documentation. After completing the tasks, 29 participants completed a survey to share their experience.

More than 80% of the participants considered the mined usage summaries as an improvement over both API official documentation and Stack Overflow, because our tool offered an increase in productivity, confidence in usage and reduction in time spent. According to one participant: *"It is quicker to find solution in [tool] since the subject is well covered and useful information is collected."* The participants considered that learning an API could be quicker while using our tool than while using official documentation or Stack Overflow, because our tool synthesizes the information from Stack Overflow by APIs using both sentiment and source code analyses.

Out of the participants, 87.1% wanted to use our tool either daily in their development tasks, or whenever they have specific needs (e.g., learning a new API). All the participants (100%) rated our tool as usable for being a single platform to provide insights about API usage and being focused towards a targeted audience. The developers praised the usability, search, and analytics-driven approach in the tool. According to one participant: *"In depth knowledge plus the filtered result can easily increase the productivity of daily development tasks, . . . with the quick glimpse of the positive and negative feedback."* As a future improvement, the developers wished our tool to mine usage scenarios from multiple online forums.

5. Threats to Validity

Internal validity warrants the presence of bias in the study. We mitigated the bias using manual validation (e.g., our benchmark datasets were assessed by multiple coders). *External validity* concerns about the *generalizability* of the results. While our evaluation corpus consists of 22.7K posts from Stack Overflow, the results will not carry the automatic implication that the same results can be expected in general. *Reliability threats* concern the possibility of replicating this study. We provide the necessary data in an online appendix [1].

6. Related Work

We discuss works related to our proposed algorithms and the overall theme of our paper, i.e., crowd-sourced documentation.

6.1. Works Related to the Three Proposed Algorithms.

As we noted in Section 1, we are aware of no techniques that can associate reactions towards code examples in forums (Section 2.5).

Our algorithm to generate summary description of tasks (Section 2.4) is different from the generation of natural language description of API elements (e.g., class [40], method [60, 61]), which takes as input source code (e.g., class names, variable names, etc.) to produce a description. We take as input the textual discussions around code examples in forum posts. Our approach is different from API review summaries [68], because our summary can contain both opinionated and neutral sentences.

Our approach to generate task description from *an answer* differs from Xu et al. [77], who proposed AnswerBot to automatically summarize *multiple answers* relevant to a developer task. The input to AnswerBot is a natural language query describing a development task. Based on the query, AnswerBot first finds all the questions in Stack Overflow whose titles closely match the query. AnswerBot then applies a set of heuristics based on Maximal Marginal Relevance (MMR) [10] to find most novel and diverse paragraphs in the answers. The final output is the ranked order of the paragaphs as a summary of the answers that could be used to complete the development task. Unlike Xu et al. [77] who focuses on the summarization of multiple answers for a given task, we focus on summarizing the contents of one answer. Unlike Xu et al. [77] who utilize only the textual contents of answers to produce the summary, we utilize both the contents from the question and answer to produce the summary. A summary of relevant textual contents from questions provides an overview of the problem (i.e., development task). Such a problem definition adds contextual information over the question title, which may not be enough to explain properly the development task. This assumption is consistent with our previous findings of surveys of software developers who reported the necessity of adding contextual and situationally relevant information into summaries produced from developer forums [67].

Our algorithm to associate a code example to an API mention in a forum post (Section 2.3) differs from the existing traceability techniques for code examples in forum posts [62, 51, 79] as follows:

- As we noted in Section 3.1, the most directly comparable to our technique is Baker [62], because both Baker and our proposed technique rely on a predefined database of APIs. Given a code example as an input, our technique differs from Baker by considering both code examples and textual contents in the forum posts to learn about which API from the API database to link to the code example. Baker does not consider textual contents in the forum posts.

- As we noted in Section 3.1, given that our technique relies specifically on an API database similar to Baker [62], our algorithm is not directly comparable to StatType as proposed by Phan et al. [51]. StatType relies on API usage patterns, i.e., how frequently a method and class name is found to be associated with an API in the different GitHub code repositories. We do not rely on the analysis of client software code to infer usage patterns of an API.

- Unlike Subramanian et al. [62, 14, 51], we can operate both with *incomplete* and *complete* API database against which API mentions can be checked for traceability. This flexibility allowed us to use an online *incomplete* API database (Maven central), instead of constructing an offline database. All the existing traceability techniques [62, 14] requires the generation of an offline *complete* API database to support traceability.

- Unlike Ye et al. [79], we link a code example in a forum post to an API mentioned in the textual contents of the forum post. Specifically, Ye et al. [79] focus on finding API methods and type names in the textual contents of forum posts, e.g., identify 'numpy', 'pandas' and 'apply' in the text 'While you can also use numpy, the documentation describes support for Pandas apply method using the following code example'. In contrast, our proposed algorithm links a provided code example with an API mentioned in the textual contents. For example, for the above textual content where Ye et al. [79] link both 'Pandas' and 'numpy' APIs, our algorithm will link the provided code example to only the 'Pandas' API.

In Section 3.1, we compared our traceability algorithm with the state of the art technique, Baker [62]. The recall of Baker was 0.49, i.e., using Baker we could not link more than 50% code examples in our evaluation - because those contained references to multiple API types/methods, but the textual contents referred to only one of those APIs. Our technique could find a link for all (i.e., 100% recall) with more than 96% precision. Our evaluation sample is statistically representative of our corpus of 8589 code examples. Therefore, using Baker we could have only found links for only 4100 of those, while our technique could link all 8589 with a very high precision. Stack Overflow contains millions of other code examples. Therefore, our technique significantly advances the state of the art of code example traceability to support task-based documentation.

Kim et al. [30] proposed FaCoy a code-to-code search engine, i.e., given as input a code snippet, the engine finds other code snippets that are *semantically* similar to the input code example. While our and FaCoY's goals remain the same, i.e., to help developers in their development tasks, we differ from each other with regards to both the outputs and the approaches. For example, given as input a code example in Stack Overflow post, we link it to an API name as mentioned in the textual contents of the post. In contrast, given as input a code example, FaCoY finds other similar code examples. Nevertheless, in the evaluation of our proposed algorithm we compared our algorithm against Google. We were able to compare Google, because given as input a code example, Google outputs links to online web sites where the

API of our interest could be cited. Nevertheless, as we noted in Section 3.1, Google search did not perform well for our particular problem. This finding is not surprising, because Google is not designed for code search, even though developers use Google for diverse development tasks which motivated us to use Google as a baseline in the first place [67]. A thorough analysis of whether and how the results from Google could be significantly improved with code preprocessing and the usage of an intermediate engine (such as FaCoY) is an interesting research question, which warrants for an extensive and stand-alone research by itself. We leave it as a future work for the software engineering research community.

6.2. Crowd-Sourced API Documentation.

The automated mining of crowd-sourced knowledge from developer forums has generated considerable attention in recent years. To offer a point of reference of our analysis of related work, we review the research papers listed in the Stack Exchange question 'Academic Papers Using Stack Exchange Data' [46] and whose titles contain the keywords ('documentation' and/or 'API') [74, 29, 59, 63, 37, 79, 8, 9, 3, 2, 75, 13, 47, 48, 28, 7, 65, 15, 33, 32]. Existing research has focused on the following areas:

- Assessing the feasibility of forum contents for documentation and API design (e.g., usability) needs,
- Answer question in Stack Overflow using formal documentation,
- Recommend new documentation by complementing both official and developer forum contents, and
- Categorizing forum contents (e.g., detecting issues).

Our work differs from the above work by proposing three novel algorithms that can be used to automatically generate task-based API documentation from Stack Overflow. As we noted in Section 1, we follow the concept of "minimal manual" which promotes task-centric documentation of manual [11, 5, 56, 36]. We differ from the above work as follows: 1. We include comments posted in the forum as reactions to a code example in our usage scenarios. 2. We automatically mine API usage scenarios from online forum, thereby greatly reducing the time and complexity to produce minimal manual.

Given the advance in techniques developed to automatically mine insights from crowd-sourced software forums, recent research on crowd-sourced API documentation has focused specifically on the analysis of quality in the shared knowledge. A number of high-impact recent research papers [81, 78, 64] warn against directly copying code from Stack Overflow, because such code can have potential bugs or misuse patterns [81] and that such code may not be directly usable (e.g., not compilable) [78, 64]. We observed both issues during the development of our proposed mining framework. We attempted to offer solutions to both issues within the context of our goal, i.e., producing task-based documentation. For example, in Section 2.2, we discussed that shared code examples can have minor syntax problem (e.g., missing semi-colon at the end of a source code line in Java), but they are still upvoted by Stack Overflow users, i.e., the users considered those code examples as useful. Therefore, to ensure such code examples can still be included in our task-based documentation, we developed a hybrid code parser that combines Island parsing with ANTLR grammar to parse code examples line by line. Based on the output of the parser, we thus can decide whether to include code example with syntax error or not. For example, if a code example has a minor error (e.g., missing semi-colon), we can decide to include it. We can, however, discard a code example that has many syntax errors (e.g., say 50% of the source code lines have some errors).

While the issues with regards to code usability in crowd-sourced code examples [78, 64] could be addressed by converting those into compilable code examples, such approach requires extensive research and technological advancement due to the diversity of such issues and the huge number of available programming languages in modern programming environment. As a first step towards making progress in this direction, in our framework, we developed the algorithm to associate reactions of other developers towards a code example. The design and development of the algorithm was motivated by our findings from previous surveys of 178 software developers [67]. The developers reported that they consider the combination of a code example and reviews about those code examples in the forum posts as a form of API documentation and they especially leverage the reviews to understand the potential benefits and pitfalls of reusing the code example.

7. Conclusions and Future Work

APIs are central to the modern day rapid software development. However, APIs can be hard to use due to the shortcomings in API official documentation, such as incomplete or not usable [54]. This resulted in plethora of

API discussions in forum posts. We present three algorithms to automatically mine API usage scenarios from forums that can be used to produce a task-based API documentation. We developed on online task-based API documentation engine based on the three proposed algorithms. Our future work focuses on the utilization of our proposed framework to improve API documentation resources, such as the development of techniques to automatically recommend fixes to common API documentation problems (e.g., ambiguity, incorrectness) [71, 54], to associate the mined usage scenarios to specific API versions, and to produce on-demand developer documentation [55]

References

[1] *Online appendix for Mining Task-Based API Documentation.* `https://github.com/anonsubmissions/ist2019`, 5 August 2019 (last accessed).

[2] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider. Classifying stack overflow posts on api issues. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 244–254, 2018.

[3] S. Azad, P. C. Rigby, and L. Guerrouj. Generating api call rules from version history and stack overflow posts. *ACM Transactions on Software Engineering and Methodology*, 25(4):22, 2017.

[4] S. Blair-Goldensohn, K. Hannan, R. McDonald, T. Neylon, G. A. Reis, and J. Reyner. Building a sentiment summarizer for local search reviews. In *WWW Workshop on NLP in the Information Explosion Era*, page 10, 2008.

[5] I. Cai. *Framework Documentation: How to document object-oriented frameworks. An Empirical Study.* PhD in Computer Sscience, University of Illinois at Urbana-Champaign, 2000.

[6] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli. Sentiment polarity detection for software development. *Journal Empirical Software Engineering*, pages 2543–2584, 2017.

[7] J. C. Campbell, C. Zhang, Z. Xu, A. Hindle, and J. Miller. Deficient documentation detection: A methodology to locate deficient project documentation using topic analysis. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*, pages 57–60, 2013.

[8] E. Campos, L. Souza, and M. Maia. Searching crowd knowledge to recommend solutions for api usage tasks. *Journal of Software: Evolution and Process*, 28(10):863–892, 2016.

[9] E. C. Campos, M. Monperrus, and M. A. Maia. Searching stack overflow for api-usage-related bug fixes using snippet-based queries. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pages 232–242, 2016.

[10] J. Carbonell and J. Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *In Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 335–336, 1998.

[11] J. M. Carroll, P. L. Smith-Kerker, J. R. Ford, and S. A. Mazur-Rimetz. The minimal manual. *Journal of Human-Computer Interaction*, 3(2):123–153, 1987.

[12] A. Cohan and N. Goharian. Revisiting summarization evaluation for scientific articles. In *Proc. Language Resources and Evaluation*, page 8, 2016.

[13] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proc. 18th Intl. Symp. Foundations of Soft. Eng.*, pages 127–136.

[14] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *34th IEEE/ACM International Conference on Software Engineering*, pages 45–57, 2012.

[15] F. Delfim and M. M. Klérisson Paixão, Damien Cassou. Redocumenting apis with crowd knowledge: a coverage analysis based on question types. *Journal of the Brazilian Computer Society*, 29(1), 2016.

[16] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proc. 34th IEEE/ACM International Conference on Software Engineering*, pages 266–276, 2012.

[17] FasterXML. *Jackson.* `https://github.com/FasterXML/jackson`, 2016.

[18] B. Fox. *Now Available: Central download statistics for OSS projects*, 2017.

[19] D. Freelon. ReCal2: Reliability for 2 coders. `http://dfreelon.org/utils/recalfront/recal2/`, 2016.

[20] D. Freelon. ReCal3: Reliability for 3+ coders. `http://dfreelon.org/utils/recalfront/recal3/`, 2017.

[21] M. Gambhir and V. Gupta. Recent automatic text summarization techniques: a survey. *Artificial Intelligence Review*, 47(1):166, 2017.

[22] Google. *Gson*. `https://github.com/google/gson`, 2016.

[23] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. D. Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proc. 35th IEEE/ACM International Conference on Software Engineering*, page 10 pages, 2013. to appear.

[24] S. G. Hart and L. E. Stavenland. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. pages 139–183, 1988.

[25] M. Hu and B. Liu. Mining and summarizing customer reviews. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 168–177, 2004.

[26] M. R. Islam and M. F. Zibran. Leveraging automated sentiment analysis in software engineering. In *Proc. 14th International Conference on Mining Software Repositories*, pages 203–214, 2017.

[27] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing source code using a neural attention model. In *In Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 2073–2083, 2016.

[28] H. Jiau and F.-P. Yang. Facing up to the inequality of crowdsourced api documentation. *ACM SIGSOFT Software Engineering Notes*, 37(1):1–9, 2012.

[29] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. Devanbu, and V. Filkov. Using and asking: Apis used in the android market and asked about in stackoverflow. In *In Proceedings of the INTERNATIONAL CONFERENCE ON SOCIAL INFORMATICS*, pages 405–418, 2013.

[30] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon. Facoy a code-to-code search engine. In *In Proceedings of the IEEE/ACM 40th International Conference on Software Engineering*, pages 946 – 957, 2018.

[31] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proc. 41st Annual Meeting on Association for Computational Linguistics*, pages 423–430, 2003.

[32] J. Li, A. Sun, and Z. Xing. Learning to answer programming questions with software documentation through social context embedding. *Journal of Information Sciences*, 448–449:46–52, 2018.

[33] J. Li, Z. Xing, and A. Kabir. Leveraging official content and social context to recommend software documentation. *IEEE Transactions on Services Computing*, page 1, 2018.

[34] B. Lin, F. Zampetti, G. Bavota, M. D. Penta, and M. Lanza. Pattern-based mining of opinions in Q&A websites. In *Proc. 41st ACM/IEEE International Conference on Software Engineering*, page 12, 2019.

[35] B. Liu. *Sentiment Analysis and Subjectivity.* CRC Press, Taylor and Francis Group, Boca Raton, FL, 2nd edition, 2010.

[36] H. V. D. Maij. A critical assessment of the minimalist approach to documentation. In *Proc. 10th ACM SIGDOC International Conference on Systems Documentation*, pages 7–17, 1992.

[37] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Hauswirth, N. Nystrom, and M. Lanza. Use at your own risk: The java unsafe api in the wild. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications*, pages 695–710, 2015.

[38] R. Mihalcea and P. Tarau. Textrank: Bringing order into texts. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 404–411, 2004.

[39] L. Moonen. Generating robust parsers using island grammars. In *Proc. Eighth Working Conference on Reverse Engineering*, pages 13–22, 2001.

[40] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for Java classes. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 23–32, 2013.

[41] N. Novielli, F. Calefato, and F. Lanubile. The challenges of sentiment detection in the social programmer ecosystem. In *Proceedings of the 7th International Workshop on Social Software Engineering*, pages 33–40, 2015.

[42] Oracle. *The Java Date API.* `http://docs.oracle.com/javase/tutorial/datetime/index.html`, 2017.

[43] S. Overflow. `http://stackoverflow.com/questions/1688099/`, 2010.

[44] S. Overflow. *Name/value pair loop of JSON Object with Java & JSNI.* `http://stackoverflow.com/questions/7141650/`, 2010.

[45] S. Overflow. *Converting JSON to Hashmap¡String, POJO¿ using GWT.* `https://stackoverflow.com/questions/20374351`, 2017.

[46] S. Overflow. *Academic Papers Using Stack Exchange Data.* `https://meta.stackexchange.com/questions/134495/academic-papers-using-stack-exchange-data`, 2019. Last accessed on 12 May 2019.

[47] C. Parnin and C. Treude. Measuring api documentation on the web. In *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, pages 25–30, 2011.

[48] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd documentation: Exploring the coverage and dynamics of api discussions on stack overflow. Technical report, Technical Report GIT-CS-12-05, Georgia Tech, 2012.

[49] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. Technical report, Georgia Tech, 2012.

[50] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 1st edition, 2007.

[51] H. Phan, H. A. Nguyen, N. M. Tran, L. H. Truong, A. T. Nguyen, and T. N. Nguyen. Statistical learning of API fully qualified names in code snippets of online forums. In *Proceedings of 40th International Conference on Software Engineering*, pages 632–642, 2018.

[52] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Prompter: Turning the IDE into a self-confident programming assistant. *Empirical Software Engineering*, 21(5):2190–2231, 2016.

[53] P. C. Rigby and M. P. Robillard. Dicovering essential code elements in informal documentation. In *Proc. 35th IEEE/ACM International Conference on Software Engineering*, pages 832–841, 2013.

[54] M. P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):26–34, 2009.

[55] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vasquez, G. C. Murphy, L. M. D. Shepherd, and E. Wong. On-demand developer documentation. In *Proc. 33rd IEEE International Conference on Software Maintenance and Evolution New Idea and Emerging Results*, page 5, 2017.

[56] M. B. Rosson, J. M. Carrol, and R. K. Bellamy. Smalltalk scaffolding: a case study of minimalist instruction. In *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pages 423–430, 1990.

[57] F. Shull, F. Lanubile, and V. R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering*, 26(11):1101–1118, 2000.

[58] P. Software. *Spring Framework*. `https://spring.io/`, 2017.

[59] L. Souza, E. Campos, , and M. Maia. On the extraction of cookbooks for apis from the crowd knowledge. In *Proceedings of the 28th Brazilian Symposium on Software Engineering*, pages 21–30, 2014.

[60] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proc. 25th IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[61] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proc. 33rd International Conference on Software Engineering*, pages 101–110, 2011.

[62] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proceedings of 36th International Conference on Software Engineering*, pages 643–652, 2014.

[63] J. Sunshine, J. D. Herbsleb, , and J. Aldrich. Searching the state space: A qualitative study of api protocol usability. In *Proceedings of the International Conference on Program Comprehension*, pages 82–93, 2015.

[64] V. Terragni, Y. Liu, and S.-C. Cheung. Csnippex: automated synthesis of compilable code snippets from qa sites. In *In Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 118–129, 2016.

[65] C. Treude and M. P. Robillard. Augmenting API documentation with insights from stack overflow. In *Proc. 38th International Conference on Software Engineering*, pages 392–403, 2016.

[66] G. Uddin, O. Baysal, and L. Guerrouj. Understanding how and why developers seek and analyze api-related opinions. *IEEE Transactions on Software Engineering*, page 13, 2017.

[67] G. Uddin, O. Baysal, L. Guerrouj, and F. Khomh. Understanding how and why developers seek and analyze API-related opinions. *IEEE Transactions on Software Engineering*, 2019.

[68] G. Uddin and F. Khomh. Automatic summarization of API reviews. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 159–170, 2017.

[69] G. Uddin and F. Khomh. Automatic summarization of api reviews. In *Submitted to 32nd IEEE/ACM International Conference on Automated Software Engineering*, page 12, 2017.

[70] G. Uddin and F. Khomh. Automatic opinion mining from API reviews from stack overflow. *IEEE Transactions on Software Engineering (TSE)*, page 36, 2019.

[71] G. Uddin and M. P. Robillard. How API documentation fails. *IEEE Softawre*, 32(4):76–83, 2015.

[72] G. Uddin and M. P. Robillard. Automatic resolution of API mentions in informal documents. In *McGill Technical Report*, page 6, 2017.

[73] J. Walnes. *Xstream*. `http://x-stream.github.io/`, 2017.

[74] W. Wang and M. W. Godfrey. Detecting api usage obstacles: A study of ios and android developer questions. In *In Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 61–64, 2013.

[75] W. Wang and M. W. Godfrey. Detecting API usage obstacles: a study of iOS and Android developer questions. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*, pages 61–64, 2013.

[76] Wikipedia. *Application programming interface*. `http://en.wikipedia.org/wiki/Application_programming_interface`, 2014.

[77] B. Xu, Z. Xing, X. Xia, and D. Lo. Answerbot: automated generation of answer summary to developers' technical questions. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 706–716, 2017.

[78] D. Yang, A. Hussain, and C. V. Lopes. From query to usable code: an analysis of stack overflow code snippets. In *In Proceedings of the 13th International Conference on Mining Software Repositories*, pages 391–402, 2016.

[79] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre. Learning to extract api mentions from informal natural language discussions. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, page 12, 2016.

[80] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *In Proceedings of the 15th International Conference on Mining Software Repositories*, pages 476–486, 2018.

[81] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim. Are code examples on an online qa forum reliable?: a study of api misuse on stack overflow. In *In Proceedings of the 40th International Conference on Software Engineering*, pages 886–896, 2018.