Mining Crowd-Sourced API Usage Scenarios

Abstract—**Developers discuss usage scenarios of APIs in the online developer forums. Unfortunately, it can be hard for a developer to find the right usage scenario for an API due to the abundance of such scenarios scattered in many posts. We present a framework to automatically mine usage scenarios about APIs. Each usage scenario of an API consists a code example, a summary description, and the reactions (i.e., positive and negative opinions) of other developers towards the code example. First, given as input a forum post that consists of a code example, we automatically link the code example to an API mentioned in the textual contents of the forum post (i.e., the code example is provided to discuss a use case of the API). Second, given as input a code example in a forum post that is linked to an API mention, we propose an adaption of the popular TextRank algorithm to automatically generate a natural language description of the code example by summarizing the usage discussions in the forum post. Third, we automatically associate all the reactions that developers posted towards the code example in the forum. We evaluate each algorithm by producing benchmark datasets. We observed a precision of 0.957 and a recall of 1.0 with the linking of a code example to an API mention in the forum. We observed a precision of 0.954 and a recall of 0.974 in the produced summaries. We observed a precision of 0.836 and a recall of 0.768 in the association of reactions to the code examples. We leveraged the framework to develop a tool that can automatically crawl Stack Overflow and produce summaries of API usage scenarios.**

Index Terms—**API, Mining, Usage, Documentation.**

I. Introduction

APIs (Application Programming Interfaces) offer interfaces to reusable software components. The learnability of an API depends on the availability and usefulness of learning resources of the API [1]–[3]. Unfortunately, despite developers' reliance on API official documentation as a major resource for learning and using APIs [4], the documentation can often be incomplete, incorrect, obsolete, and not usable [5].

The shortcomings in the official documentation led to the creation and popularity of online developer forums (e.g., Stack Overflow), where developers can ask and answer questions on how to address diverse development tasks that may involve APIs. A number of research efforts have focused on integrating information from the forum posts into API official documentation, such as the linking of API types in Javadocs to code examples in forum posts [6], the presentation of interesting textual contents from Stack Overflow about an API type in Javadocs [7], etc. (see Section II).

While the linking of a code example to an API type in the Javadoc offers greater insights into how the API type can be used than the actual API official documentation, this approach can be infeasible in the following scenarios: **(1)** When the code examples may be merely using the API type as a utility, but instead the development tasks in the code examples are *more specific* to another API. **(2)** When the usage of a development task may require more than the linking to a code example, such as contextual information on how to use it.

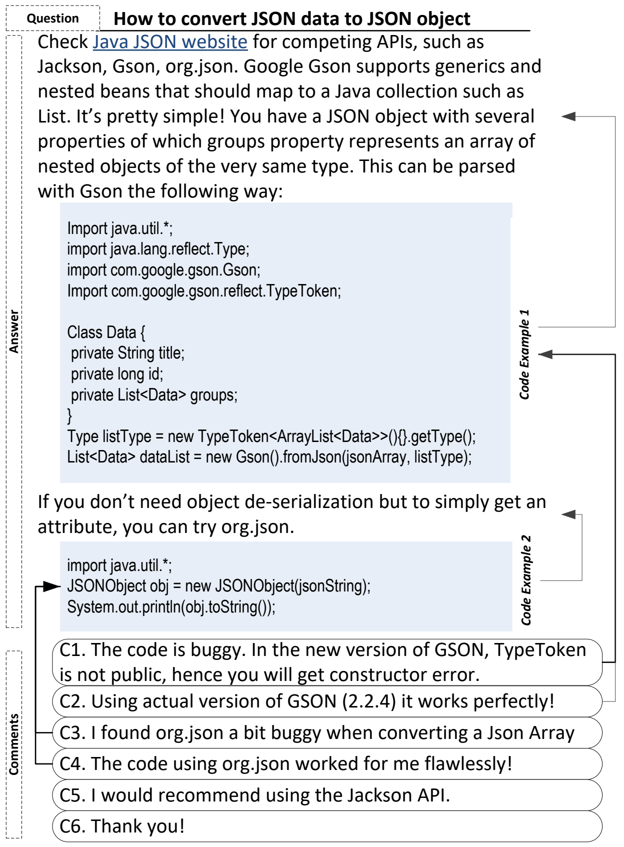
As a demonstration of the first problem, consider the two usage scenarios presented in Figure 1. Both of the two usage scenarios use multiple types and methods from the java.util API. In addition, the first code example uses the java.lang API. However, both scenarios are related to conversion of JSON data to JSON object. As such, the two usage scenarios introduce two open source Java APIs to complete the task (Google GSON in scenario 1 and org.json in scenario 2). This focus is easier to understand when we look at the textual contents that describe the two usage scenarios.

As a demonstration of the second problem, consider the reactions to the two usage scenarios in the comments to the answer in Figure 1. 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 usage scenario for GSON is not buggy in the newer version of the 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 usage scenario 2 (i.e., the one related to org.json API) works flawlessly. Indeed, reviews about APIs can be useful to learn about specific nuances and use cases of the provided code examples [8], [9].

We present a framework to automatically mine *API usage scenarios* from developer forum posts. Each scenario consists of four items: **(1)** A code example as discussed in a forum post, **(2)** An API about which the code example is provided to address a development task (for example, the API GSON for Code Example 1 in Figure 1), **(3)** A natural language description of the code example to summarize what the code does, and **(4)** A list of reactions as positive or negative opinions from other developers towards the code example.

Our proposed API usage scenario mining framework works as follows. First, given as input a forum post that consists of a code example, we automatically link the code example to an API mentioned in the textual contents of the forum post (i.e., the code example is provided to discuss a use case of the API). Second, given as input a code example in a forum post that is linked to an API mention, we propose an adaptation of the popular TextRank algorithm [10] to automatically generate a natural language description of the code example by summarizing the usage discussions in the forum post. Third, we automatically associate all the reactions that developers posted towards the code example in the forum.

We evaluated our framework by producing three benchmarks, one each for the three algorithms. We observed preci-



How to convert JSON data to JSON object

Check Java JSON website for competing APIs, such as Jackson, Gson, org.json. Google Gson supports generics and nested beans that should map to a Java collection such as List. It's pretty simple! You have a JSON object with several properties of which groups property represents an array of nested objects of the very same type. This can be parsed with Gson the following way:

```
Import java.util.*;
import java.lang.reflect.Type;
import com.google.gson.Gson;
Import com.google.gson.reflect.TypeToken;

Class Data {
 private String title;
 private long id;
 private List<Data> groups;
}
Type listType = new TypeToken<ArrayList<Data>>(){}.getType();
List<Data> dataList = new Gson().fromJson(jsonArray, listType);
```

If you don't need object de-serialization but to simply get an attribute, you can try org.json.

```
import java.util.*;
JSONObject obj = new JSONObject(jsonString);
System.out.println(obj.toString());
```

C1. The code is buggy. In the new version of GSON, TypeToken is not public, hence you will get constructor error.
C2. Using actual version of GSON (2.2.4) it works perfectly!
C3. I found org.json a bit buggy when converting a Json Array
C4. The code using org.json worked for me flawlessly!
C5. I would recommend using the Jackson API.
C6. Thank you!

Fig. 1. How API usage scenarios are discussed in forum posts.

sions of 0.957, 0.954, and 0.836 and recalls of 1.0, 0.974, and 0.768 with the linking of a code example to an API mention in the forum posts, the produced summaries, and the association of reactions to the code examples, respectively. We compared the algorithms against a total of eight state of the art baselines. Our algorithms outperformed all the baselines.

Contributions. We make the following major contributions:

- **Framework.** We propose a framework to automatically mine API usage scenarios from developer forum posts. We present three algorithms to 1) associate a code example with an API mention in the forum posts, 2) generate natural language summary description of the code example, and 3) associate reactions related to the code example as observed in the forum. We incorporate the framework in our tool to search and summarize API usage.
- **Evaluation.** We create three benchmarks and compare the algorithms using the benchmarks against eight baselines.

We advance the state of the art techniques by 1) Complementing the code example to API linking algorithms in developer forums [6], [11] by proposing an algorithm to associate the code example to an API mentioned in the forum texts. 2) Proposing an algorithm to associate code examples to the reactions of other developers and an adaptation of the TextRank algorithm to automatically generate natural language description of code examples in forum posts.

II. Conception of the Mining Framework

In this section, we explain the design decisions behind our proposed framework by corroborating it with as well as differentiating it from the selected work in the literature.

• **Task-oriented API Documentation.** As we noted in Section I, the focus of this paper is to establish a framework to automatically mine usage scenarios about APIs from developer forums that can facilitate task-orientation documentation for APIs. We follow the concept of "minimal manual" which promotes task-centric documentation of manual. The format is proven to work better than the traditional API documentation [12]–[15]. In a subsequent study, Shull et al. [16] examined 43 participants and observed that the participants abandoned hierarchy-based documentation (e.g., Javadocs) in favor of the example applications based documentation, because it took them longer to start writing application using the hierarchy-based documentation.

To offer a task-based documentation experience using our proposed framework, 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 with the code example a short text describing how the task is supported by the code example. We differ from the above work as follows: 1) We include comments as posted in a forum post as reactions to a code example in our usage scenarios. 2) We present a framework to automatically mine such usage scenarios from online forum posts, thereby greatly reducing the time and complexity that may be required to produce those manually.

• **Code Example to API Traceability.** One of the three algorithms in our mining framework is the automatic association of a code example in a forum post to an API about which the code example is provided. Our algorithm borrows concepts from the existing traceability techniques [6], [17]: we need to generate fully qualified type names in the code examples. Our algorithm also differs from the above work as follows: We detect an API as an API name as mentioned in the textual contents in the forum posts. We propose techniques to automatically link the API mention to a code example. This design decision allows us to 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" [18]. For example, in Figure 1, we consider the followings as APIs: 1) Google GSON, 2) Jackson, 3) org.json, 4) java.util, and 5) java.lang. Each API thus can contain a number of modules (e.g., packages) and elements (e.g., API types, such as class, interface, 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 [19]). As we observe in Figure 1, this is also how APIs can be mentioned in online forum posts.

In addition, our algorithm differs the related work as follows: 1) Unlike [6], [11], [17], we can operate both with *incomplete* and *complete* API database against which API mentions can be checked for traceability. An API database in our algorithm can be incomplete, if it has only type names of an API instead of both type and method names (and the structural relationships between the types and methods). This flexibility allowed us to use on online *incomplete* API database (Maven central), instead of constructing an offline database. All the existing traceability techniques [6], [11] requires the generation of an offline *complete* API database to support traceability. 2) Unlike [17], we do not rely on the analysis of client software code to infer usage patterns of an API. While such analysis can offer better accuracy than technique [6] using API databases, the approach can be infeasible when such client software code may not be available (e.g., for a new API). 3) Unlike all the above work, we derive a *hybrid* parser that can tolerate minor syntactical error in the code examples from forum posts. As we demonstrate in Section III and Section **??**, a code example shared in a forum post can have minor syntax errors, but it can still be considered as useful by other developers in the forums.

• **Mining Usage Scenarios from Forums Vs. Execution Traces.** Dynamic analysis is used on the execution traces to detect usage patterns [20]–[23]. The techniques instrument software and collect execution traces by running the software for relevant task scenarios. Static analysis is used on the version control data of software to find API usage concepts [24]–[28]. To the best of our knowledge, ours is the first technique to mine API usage scenarios from developer forums by combining code examples and developer reactions. We parse code examples using static analyses, so we do not require instrumented client software.

• **Summary Description of API Usage Scenarios.** Our algorithm to generate summary description of code examples in forum posts is different from the generation of natural language description of API elements (e.g., class [29], method [30], [31]), which take as input source code (e.g., class names, variable names, etc.) to produce a description. Our approach is also different from API review summaries [8], because we do not categorize opinionated sentences into aspects.

III. Algorithms of the Mining Framework

Our mining framework takes as input a forum post and outputs all the usage scenarios found in the post. For example, given as input the forum post in Figure 1, the framewrok returns two usage scenarios: (1) The code example 1 by associating it to the API Google GSON, the two comments (C1, C2) as reactions, and a summary description of the code example in natural language to inform the specific development task that can be addressed using 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 2):

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.

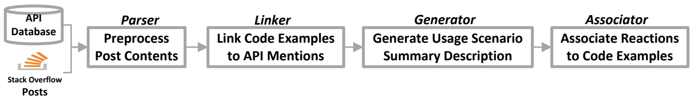

API Database
Stack Overflow Posts
Parser
Preprocess Post Contents
Linker
Link Code Examples to API Mentions
Generator
Generate Usage Scenario Summary Description
Associator
Associate Reactions to Code Examples

Fig. 2. The major components of our API usage scenario mining framework

4) A **Generator** to produce a natural language summary description of a code example
5) An **Associator** to find reactions towards code examples.

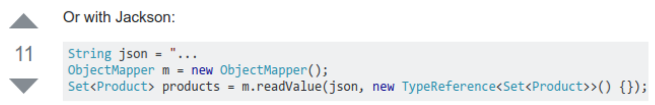
A. API Database

An API database is required to infer the association between a code example and an API mentioned in the forum texts. Our API 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 source code packages. Each package can have one or more code elements, such as classes, methods, etc. As noted in Section II, for the Java official APIs available through the Java SDKs, we consider an official Java package as an API. For each API in our database, we record the following meta-information: (1) the name of the API, (2) the URLs to the different resources of the API (e.g., the official documentation), (3) the dependency of the API on other APIs, (4) the names of the modules of the API, (5) the package names under each module, (6) the type names under each package, and (7) 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 four items from the pom.xml files of the open-source APIs hosted in online Maven Central repository. All of the APIs hosted in Maven central are for Java. We picked the Maven Central because it is the primary source for hosting and searching for Java APIs with over 70 million downloads every week [32].

B. Preprocessing of Forum Posts

Given as input a forum post, we preprocess its content as follows: (1) We categorize the post content into three types: (a) *code snippets*; (b) *hyperlinks*; and (c) *natural language text* representing the rest of the content. (2) We detect individual sentences in the *natural language text*. (3) Following Dagenais and Robillard [11], we discard the following *invalid* code examples during our parsing: (a) Non-code snippets (e.g., XML, JSON extract), (b) Non-Java snippets (e.g., JavaScript). The rest of the code examples are considered as *valid*.

• **Hybrid Code Parser.** We parse each valid code snippet using a hybrid parser combining ANTLR [33] and Island Parser [34]. 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 3. An ANTLR Java parser fails to parse it 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


Or with Jackson:

11

```
String json = "...
ObjectMapper m = new ObjectMapper();
Set<Product> products = m.readValue(json, new TypeReference<Set<Product>>() {});
```

Fig. 3. A popular code example with a syntax error (Line 1) [35]

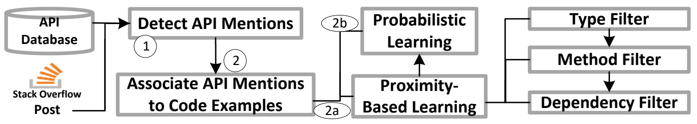


Fig. 4. The major components to link a code example to API mention

lines. For this paper, we focused only on Java code examples. Therefore, we use semi-colon as the line separator indicator. 2) We attempt to 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) [11], [36]. We collect types that are most likely not declared by the user. Consider the first code example in Figure 1. We add `Type`, `Gson` and `TypeToken` into our code context, 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 a separate post can be used in another post without any explicit mention of the code type. For example, consider the example:

```
ObjectMapper mapper = new ObjectMapper();
Wrapper = mapper.readValue(jsonStr , Wrapper.class);
```

We associate the `mapper` object to the `ObjectMapper` type.

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

C. 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 4):

Step 1. Detect API Mentions: We detect API mentions in the textual contents of forum posts following Uddin and Robillard [37]. 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 [37], we apply both exact and fuzzy matching. For example, for API mention 'Gson' in Figure 1, 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 5, we show a partial Mention Candidate List for the mention 'gson'. Each rounded rectangle denotes an API candidate with its name at the top and one or more module names at the bottom (if module names matched). In reality, this list contains 129 APIs.

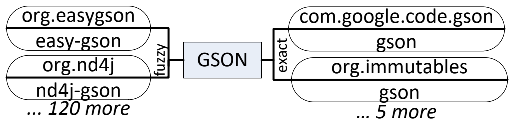


Fig. 5. Partial Mention Candidate List of GSON in Figure 1

```
input  : (1) Code Example $C = (T, E)$, (2) API
         Mentions in buckets $B = (B_b, B_a, B_t)$
output: Association decision, $D = \{d_{mention}, d_{api}\}$
1  Proximity Filters $F = [F_{type}, F_{method}, F_{dep}]$;
2  $D = \emptyset$, $N$ = length($B$), $K$ = length($F$);
3  for $i \leftarrow 1$ to $N$ do
4      $B_i = B[i]$, $H$ = getMentionApiTuples ($B_i$);
5      for $k \leftarrow 1$ to $K$ do
6          Filter $F_k = F[k]$, $H$ = getHits ($F_k$, $C$, $H$, $L_i$);
7          if $|H| = 1$ then $D = H[1]$; break;
8  procedure getMentionApiTuples($B$)
9      List< MentionAPI > $M = \emptyset$;
10     foreach Mention $m \in B$ do
11         $MCL = \{a_1, a_2, \ldots a_n\}$        ▷ MCL of $m$;
12         foreach API $a_i \in MCL$ do
13             MentionAPI ma = $\{m, a_i\}$; $M$.add (ma)
14     return $M$;
15 procedure getHits ($F_k$, $C$, $H$)
16     $S = \emptyset$;
17     for $i \leftarrow 1$ to length ($H$) do
18         $S[i]$ = compute score of $H[i]$ for $C$ using $F_k$;
19     if max ($S$) = 0 then return $H$;
20     else
21         $H_{new} = \emptyset$;
22         for $i \leftarrow 1$ to length ($H$) do
23             if $S[i]$ = max ($S$) then $H_{new}$.add ($H[i]$);
24         return $H_{new}$
25 return $D$
```

Algorithm 1: Associate a code example to an API mention using proximity learning

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 in the same post. 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 with 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 *probabilist-learning*, i.e., by assigning the code example to an API that could be the most likely based on the observations in other forum 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 as a tuple of the API types and methods parsed from 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 name as found in the forum text (e.g., Google GSON for the first code example in Figure 1) 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. For example, for the first code example in Figure 1 the output is (Google GSON, `com.google.code.gson`).

The algorithm uses three filters (L1, discussed below). Each filter takes 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* [6]. 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 name) in the code example, we search for its occurrence in the candidate APIs. Each candidate API is an API in the list of mention API tuples. 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. When no types were matched between a candidate and the snippet, we assign Types(c_i) = $\emptyset$, i.e., $|\text{Types}(s_i) \bigcap \text{Types}(c_i)| = 0$. 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:

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

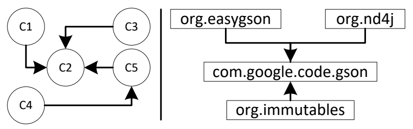


Fig. 6. Dependency graph given a hit list

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

F3. Dependency Filter. For the APIs in a hit list, we create a *dependency graph* by consulting the dependency list of each API in the database. Each node in the graph corresponds to an API from the hit list. An edge is established between two APIs in the graph, if one of the APIs dependent on the other API. From this graph, we find the API with the maximum number of incoming edges, i.e., the API on which most of the other APIs in the hit list 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 6, we show an example dependency graph (left) and a partial dependency graph for the four candidate APIs in the mention candidate in Figure 5 (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 1:

```
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 5: org.immutables and com.google.code.gson. However, as we see in Figure 6 (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 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 among developers, developers tend to refer to it in the code examples without mentioning the API (see for example [38]).

We thus associate a code example to an API which is most frequently associated to the types found in the code example in other forum posts. Given as input a code snippet S_i, the rule works in the following steps:

1) For each type T_i in S_i, we determine how frequently T_i appeared in other already associated code snippets A. We create a matrix M as $C \times T$ where each row corresponds to an API C_i found in the already associated code snippets. Each column corresponds to a type T_i in S_i. We only add an API C_i in the matrix if at least one of the types in

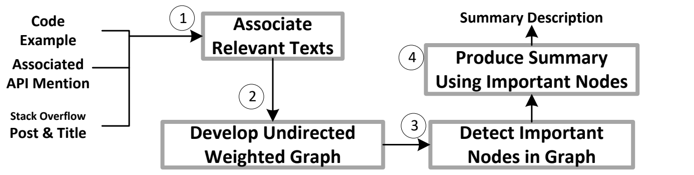

Code Example
Associated API Mention
Stack Overflow Post & Title
1
Associate Relevant Texts
2
Develop Undirected Weighted Graph
3
Detect Important Nodes in Graph
4
Produce Summary Using Important Nodes
Summary Description

Fig. 7. The major steps to produce summary description of a code example associated to an API mention

T was linked to the API in the already associated code snippets. Each co-ordinate in the matrix for a given API C_i and type T_i contains the frequency $freq(T_i, C_i)$.

2) For the given input snippet S_i, we compute the coverage of each API C_i in the matrix M. We associate the snippet S_i to the API C_i with the maximum coverage value.

D. Generating Summary Description of Usage Scenario

Our algorithm is based on the TextRank algorithm [10]. The TextRank algorithm is unsupervised. It is based on Google PageRank algorithm. TextRank is proven to produce high quality summary from an input natural language text [10], [39]. We produce summary description only for code examples that are found in the answers to questions. This is based on the observation that such a code example is in more need to be understood within the context of a development task [6]. Our algorithm operates in four steps (Figure 7):

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 which is associated with the code example. For example, for the first code example in Figure 1 which is associated 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 generate next possible sentence from this sentence by looking for two types of signals: a) the sentence refers to the API mentioned in the sentence (e.g., using a pronoun), and b) the sentence refers to a feature discussed in the sentence. To identify features, we use noun phrases. To detect noun phrases, we use shallow parsing [40]. 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. In a traditional developer forum, such as Stack Overflow, this corresponds to the title of thread where the code example if found. 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 of both 'Problem' and 'Solution' 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. In particular, we sought to find optimal weight for each node in the graph by repeatedly iterating over the following equation (until no further optimizations are possible):

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

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.** We rank the nodes by their weights, with the node with the highest weight at the top. We pick the top N nodes. We rank the nodes based on their appearance in the original post. Each node corresponds to a sentence in the post. We output the summary by combining all the ranked and ordered sentences and precede those with the title of the post. We produce a summary description by combining the three items, i.e., Title, Problem, and Solution.

E. Associating Reactions to Usage Scenarios

We associate the positive and negative opinions to a code example by following the principles of discourse learning [41]. The inputs to the algorithm are all the comments towards the post where the code example is found. The output is a list of opinionated sentences that are related to the code example. Each opinionated sentence contains positive or negative sentiments towards the code example or the API that is associated to the code example. For example, for the first code example in Figure 1, the related reactions are the first two comments (C1, C2). 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 1, 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. We detect opinionated sentences following Uddin and Khomh [8], which we found to be a better performer (with more than 0.73 precision and recall) to detect opinions in Stack Overflow.

TABLE I
DESCRIPTIVE STATISTICS OF THE DATASET (WITH ONLY 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

IV. EVALUATION OF THE MINING FRAMEWORK

Our investigation focused on the feasibility of our mining framework. We answer three research questions: **(RQ1)** What is the performance of linking scenario to API mention? **(RQ2)** What is the performance of the generation of natural language summary description of scenario? **(RQ3)** What is the performance of linking reactions to 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. We report using four performance measures: precision (P), recall(R), F-measure ($F1$), and Accuracy (A).

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

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

Evaluation Corpus. We collect all the code examples and textual contents found in the Stack Overflow threads tagged as 'Java+JSON', i.e., the threads contained discussions and opinions related to the json-based software development tasks using Java APIs. This dataset was previously used by Uddin and Khomh [8] to summarize reviews about APIs. They observed that this dataset offers a rich set of competing APIs with diverse usage discussions. We selected the Java JSON-based APIs because JSON-based techniques can be used to support diverse development scenarios, such as, both specialized (e.g., serialization) as well as utility-based (e.g., lightweight communication), etc. Our dataset was constructed from Stack Overflow 2014 data dump, which allowed us to also check whether the API official documentation could have been updated recently with scenarios that are discussed in 2014. Recent studies [6], [17] used Stack Overflow 2013 dataset.

In Table I we show descriptive statistics of the dataset. There were 22,733 posts from 3,048 threads with scores greater than zero. Following [8], we did not consider any post with a negative score because such posts are considered as not helpful by the developers. 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. On average, around four users participated in one thread, and more than one user participated in 2,940 threads (96.4%), and a maximum of 56 distinct users participated in one thread [42].

TABLE II
ANALYSIS OF AGREEMENT AMONG THE CODERS TO VALIDATE THE ASSOCIATION OF APIS TO CODE EXAMPLES (USING RECAL3 [44])

	Kappa (Pairwise)	Fleiss	Percent	Krippendorff's α
Overall	96.6	96.5	99.35	96.55
Valid	93.2	93	98.7	93.1
Discarded	100	100	100	100

Threats to Validity. *Internal validity* warrants the presence of systematic error (e.g., bias) in the study. We mitigated the bias using manual validation (e.g., evaluation corpus was assessed by three coders). *External validity* concerns about the *generalizability* of the results. While our dataset consists of hundreds of APIs from more than 22K 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 details and data in online appendix [43].

V. RESULTS

In this section, we answer the research questions.

RQ_1: Performance of Linking Code Example to API Mention

• ***Approach**:* We assess the performance of our algorithm to link a code example to an API mention using a benchmark. The benchmark consists of randomly selected 730 code examples from our entire dataset. 375 code examples were sampled from the list of 8589 valid code snippets and 355 from the list of 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 (with a 95% confidence interval). The evaluation corpus was manually validated by three independent coders: The first two coders are the first two authors of this paper, respectively. The third coder is a graduate student and is not a co-author of this paper. The benchmark creation process involved three steps:

Step 1. **Agreement Set.** 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 invalidated code examples. The two types of code examples are detected using the invalid code detection component, which as noted before, is adapted from [11].

Step 2. **Agreement Calculation.** The overall agreement among the coders was near perfect (Table II): pairwise Cohen κ was 96.6%, the percent agreement was 99.35% and Krippendorff's α was 96.5%. To resolve disagreements on a given code example, we took the majority vote.

Step 3. **Complete Coding.** Since the agreement level was near perfect, we considered that if any of the coders does further manual labeling, there is little chance that those labels would introduce any subjective bias in the assessment. The first coder then labeled the rest of the code examples. The manual assessment found nine code examples as invalid. For each of those code examples, we labeled our algorithm as wrong, i.e.,

TABLE III
PERFORMANCE OF ASSOCIATION OF CODE EXAMPLES TO API MENTIONS

Proposed Algorithm	**Precision**	**Recall**	**F1 Score**	**Acc**
Detect Invalid	-	-	-	0.969
Link to Valid w/Partial info	0.938	1.0	0.968	0.938
Link to Valid w/Full info	0.957	1.0	0.978	0.957
Overall w/Partial Info	0.938	0.969	0.953	0.953
Overall w/Full Info	0.957	1.0	0.978	0.957
Baselines (applied to valid code examples)				
B1. Baker	0.973	0.486	0.646	0.477
B2. Search (Google)	0.389	0.881	0.539	0.369

false positives. In the end, the benchmark consisted of 367 valid and 364 invalid code examples.

Baselines. We compare our algorithm against two baselines on the valid code examples.

B1. Baker: As we noted in Section II, related techniques [6], [11], [17] find full qualified names of the API elements in the code examples. Therefore, if a code example contains code types from multiple APIs, the techniques link the code example with all of those APIs. As we explained in Section I (Figure 1), this approach may not be optimal to link a code example to an API mention about which the code example is discussed in a forum post. We compare our algorithm against Baker as follows. a) Code example consisting of code Types Only from one API: We attempt to link it using the technique proposed in Baker. b) Code example consisting of code types from more than one API: if the code example is associated to one of the API mentions in the post, we leave it as 'undecided' by Baker. We do not penalize Baker in such cases, because Baker is not designed to link a code example to an API mention. 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 [17]).

B2. Search: Because search engines are predominantly used in development activities involving API usage [45], we search each code example in the online search engine Google. While assessing the search results, we check the first three hits (without advertisement). If at least one of the hits contains a reference to the associated API, we label the result as correct. Otherwise we label it as wrong. We do not consider a search result as relevant if it points to the same Stack Overflow post where the code example is found.

• ***Results**:* We achieved a precision of 0.957 and a recall of 1.0 using our algorithm (see Table III). A recall of 1.0 was achieved due to greedy approach of algorithm to attempt to find an association for each code example. The baseline Baker shows the best precision among all (0.973), but with the lowest recall (0.486). This level of precision corroborates the precision reported by Baker on Android SDKs [6]. The low recall is due mainly 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. The Google search results show the lowest precision (0.389), confirming the assumption that Google is primarily a generally purpose search engine.

We report the performance of our algorithm under different settings: 1) **Detect Invalid.** We observed an accuracy of 0.969 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 0.938 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 [6], [11], [17]. 3) **Link to valid with Full Info.** When we used all the filters under proximity learning, the precision level was increased to 0.957 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 [6], [11]. 4) **Overall.** We achieved an overall precision of 0.938 and a recall of 0.969 in our entire benchmark 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:       , dataType:       })
```

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 [46] 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 the mentions of both the Google GSON and the `com.google.gwt` API. However, `com.google.gwt` was mentioned using an acronym GWT and the API mention detector missed it.

```
String serializeToJson(Test test) {
AutoBean<Test> bean = AutoBeanUtils.getAutoBean(test);
return AutoBeanCodex.encode(bean).getPayload();}
```

RQ$_2$: Performance of the Summary Description Algorithm

• ***Approach**:* The evaluation of natural language summary description can be conducted in two ways [47]: 1) User study: participants are asked to rate the summaries 2) Benchmark: The summaries are compared against a benchmark. In benchmark-based settings, produced summaries are compared against those in the benchmark using different metrics, e.g., coverage and precision of the sentences in the summaries.

In our previous benchmark (RQ$_1$.1), a total of 125 code examples were found in the answer posts, i.e., each code example is provided in an attempt to suggest a solution to a development task. We assess the performance of our summarization algorithm for those code examples. We create another benchmark by manually producing summary description for

TABLE IV
PERFORMANCE OF ALGORITHMS TO PRODUCE SUMMARY DESCRIPTION

Techniques	Precision	Recall	F1 Score	Acc
Proposed Algorithm	0.954	0.974	0.950	0.963
Baselines				
B1. Luhn	0.830	0.796	0.804	0.819
B2. Textrank	0.830	0.796	0.804	0.819
B3. Lexrank	0.830	0.796	0.804	0.819
B4. LSA	0.831	0.801	0.808	0.822

the 125 code examples. To produce the summary description, we used 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. On average, each summary contains 7.8 sentences and 152.8 words.

Baselines. We compare the summarization algorithm against four off-the-shelf extractive summarization algorithms [48]. 1) Luhn, 2) Lexrank, 3) TextRank, and 4) Latent Semantic Analysis (LSA) The first three algorithms were previously used to summarize API reviews [8]. The LSA algorithms are commonly used in information retrieval and software engineering both for text summarization and query formulation [49]. Extractive summarization techniques are the most widely used automatic summarization algorithms [48]. Our proposed algorithm utilizes the TextRank algorithm. Therefore, by applying the TextRank algorithm without the adaption we proposed, we can estimate the impact of the proposed changes in algorithm.

• ***Results:*** We achieved the best precision (0.954) and recall (0.974) 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 0.830 and a recall of 0.796 (i.e., without the improvement 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 the TextRank algorithm on the Problem and Solution text blocks. This approach was necessary, because the baseline algorithms showed lower recall due to their selection of un-informative texts. Out of the four baselines, the LSA algorithm shows the best performance.

RQ$_3$: Performance of Linking Reactions to Code Examples

• ***Approach:*** We assess the performance of our algorithm to link reactions to a code example using a benchmark. The benchmark is produced by manually associating reactions to each of the 125 code examples that we used in RQ$_{1.2}$. A reaction is a positive or a negative comment towards a target entity. An entity can be the code example, the API mention which is associated to the code example, or another API that may not be related to the code example. We produced the benchmark as follows. 1) For each code example, we find the post id in Stack Overflow. Each post is an answer. 2) For each answer post, we collect all the comments that are posted in response to the answer. 3) For each comment, we label its polarity (positive, negative, or neutral) manually. 4) For each comment, we detect API mentions in the comment and assign those to the most probable API. 5) For each reaction, we label it either 1 or 0. A label of 1 indicates that the comment is related to the code example and a 0 otherwise. In total, there were 493 comments posted in response to the 125 answers in the benchmark. Out of the 493 comments, 245 are labeled as reactions (average 3.6 per answer post).

TABLE V
PERFORMANCE OF ASSOCIATING REACTIONS TO CODE EXAMPLES

Technique	Precision	Recall	F1 Score	Acc
Proposed Algorithm	0.836	0.768	0.794	0.878
B1. All Comments	0.480	0.791	0.565	0.480
B2. All Reactions	0.689	0.703	0.678	0.793

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 may work, i.e., when we do not consider the proximity of the reactions to the code example. The second baseline offers us insights on whether the simple reliance on sentiment detection would suffice.

• ***Results:*** We observed the best precision (0.836) using our proposed algorithm and the best recall (0.791) using the baseline All Comments. 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 in hand, such discussions are not relevant (e.g., when the code example offers a complete solution). 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 code example, such reactions are also irrelevant.

VI. CONCLUSIONS

Developers discuss API usage scenarios in forum posts. Due to the plethora of APIs discussed in forum posts, it can be challenging to get a quick and informed insights from forums about APIs. We present a framework to automatically mine API usage scenarios from forums. To assist developers in their development task completion using the mined scenarios, we developed on online search and summarization engine for API usage scenarios. Our future work will focus on the further utilization of the mined scenarios to research improvement opportunities in API learning and documentation resources, such as the development of tools and techniques to automatically recommend fixes to common API documentation problems (e.g., ambiguity, incorrectness, etc.) [1], [5] and to produce on-demand development documentation [50].

References

[1] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 26–34, 2009.

[2] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[3] J. Stylos, "Making APIs more usable with improved API designs, documentations and tools," PhD in Computer Sscience, Carnegie Melon University, 2009.

[4] 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*, vol. 21, no. 5, pp. 2190–2231.

[5] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Softawre*, vol. 32, no. 4, pp. 76–83, 2015.

[6] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proc. 36th International Conference on Software Engineering*, 2014, p. 10.

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

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

[9] ——, "Opiner: A search and summarization engine for API reviews," in *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 978–983.

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

[11] B. Dagenais and M. P. Robillard, "Recovering traceability links between an API and its learning resources," in *Proc. 34th IEEE/ACM Intl. Conf. on Software Engineering*, 2012, pp. 45–57.

[12] J. M. Carroll, P. L. Smith-Kerker, J. R. Ford, and S. A. Mazur-Rimetz, "The minimal manual," *Journal of Human-Computer Interaction*, vol. 3, no. 2, pp. 123–153, 1987.

[13] I. Cai, "Framework documentation: How to document object-oriented frameworks. an empirical study," PhD in Computer Sscience, University of Illinois at Urbana-Champaign, 2000.

[14] 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*, 1990, pp. 423–430.

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

[16] F. Shull, F. Lanubile, and V. R. Basili, "Investigating reading techniques for object-oriented framework learning," *IEEE Transactions on Software Engineering*, vol. 26, no. 11, pp. 1101–1118, 2000.

[17] 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*, 2018, pp. 632–642.

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

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

[20] D. Lo and S. Maoz, "Mining hierarchical scenario-based specifications," in *Proc. IEEE/ACM international conference on Automated software engineering*, 2009, pp. 359–396.

[21] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery," in *Proc. 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 460–469.

[22] D. Fahland, D. Lo, and S. Maoz, "Mining branching-time scenarios," in *Proc. IEEE/ACM international conference on Automated software engineering*, 2013, pp. 443–453.

[23] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," in *Proc. 14th IEEE International Conference on Program Comprehension*, 2006, pp. 84–88.

[24] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal API usage patterns," in *Proc. 26th IEEE/ACM Intl. Conf. on Automated Software Engineering*, 2011, pp. 456–459.

[25] ——, "Temporal analysis of API usage concepts," in *Proc. 34th IEEE/ACM Intl. Conf. on Software Engineering*, 2012, pp. 804–814.

[26] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 11–20.

[27] M. P. Robillard and B. Dagenais, "Recommending change clusters to support software investigation: An empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 143–164, 2010.

[28] M. P. Robillard, "Representing concern in source code," PhD in Computer Science, University of British Columbia, 2003.

[29] 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*, 2013, pp. 23–32.

[30] 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*, 2010, pp. 43–52.

[31] 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*, 2011, pp. 101–110.

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

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

[34] L. Moonen, "Generating robust parsers using island grammars," in *Proc. Eighth Working Conference on Reverse Engineering*, 2001, pp. 13–22.

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

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

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

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

[39] A. Nenkova and K. McKeown, "Automatic summarization," *Journal of Foundation and Trends in Information Retrieval*, vol. 5, no. 2–3, pp. 103–233, 2011.

[40] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proc. 41st Annual Meeting on Association for Computational Linguistics*, 2003, pp. 423–430.

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

[42] S. Overflow, *A better Java JSON library*, http://stackoverflow.com/questions/338586/, 2010, (last accessed in 2014).

[43] *Mining Automatic API Usage Scenarios*, https://github.com/anonsubmissions/icse2019, 24 August 2018 (last accessed).

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

[45] 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*, 2012, pp. 266–276.

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

[47] A. Cohan and N. Goharian, "Revisiting summarization evaluation for scientific articles," in *Proc. Language Resources and Evaluation*, 2016, p. 8.

[48] M. Gambhir and V. Gupta, "Recent automatic text summarization techniques: a survey," *Artificial Intelligence Review*, vol. 47, no. 1, p. 166, 2017.

[49] 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*, 2013, p. 10 pages, to appear.

[50] 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*, 2017, p. 5.