Uncovering Library Features from API Usage on Stack Overflow

Camilo Velázquez-Rodríguez
Vrije Universiteit Brussel
Brussels, Belgium
camilo.ernesto.velazquez.rodriguez@vub.be

Eleni Constantinou
Eindhoven University of Technology
Eindhoven, The Netherlands
e.constantinou@tue.nl

Coen De Roover
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

***Abstract*—Selecting an appropriate library for reuse within a vast software ecosystem can be a daunting task. A list of features for each library, i.e., a short description of the functionality that can be reused with code examples that illustrate its usage, may alleviate this problem. In this paper, we propose a data-driven approach that uses both the code snippets and the accompanying natural language descriptions from Stack Overflow posts to produce a list of features of a given library. Each extracted feature corresponds to a cluster of API classes and methods considered related based on attributes of the Stack Overflow posts in which they appear. We evaluated the approach considering seven Maven libraries and compared the resulting features against library descriptions from cookbook-like tutorials. The approach achieves an average accuracy of 67% across the seven libraries for the tutorial-like features. For at least 73% of the features extracted by the approach but missing from the documentation, we found a matching library usage in a corpus of GitHub projects. These results suggest that our clusters represent library features, which paves the way to better tool support for documenting software libraries and for selecting a library in an ecosystem.**

***Index Terms*—features; libraries; stack overflow**

I. Introduction

It is common in contemporary software development to reuse functionality from third-party libraries in order to reduce development time and potentially improve overall system quality [1]. Each library targets a particular domain (e.g., graphical user interface, persistence) and offers functionality to client systems through its Application Programming Interface (API) that facilitates implementing a particular task (e.g., displaying a dialog, serialising to JSON).

Software ecosystems such as Maven or NPM host an ever-increasing number of software libraries[1]. As the number of libraries grows, some are bound to offer similar features. For example, Maven currently has 72 different libraries that provide reusable implementations of collection data types[2]. From the brief description of each library, one can infer that they provide different kinds of collections (e.g., maps, sets, queues, stacks, etc.) but the range of features offered is not immediately clear (e.g., persistent or thread-safe collections, specific utility methods for sorting or reversing collections), neither how these features can be used (e.g., a single static method, an instance method of a class that needs to be instantiated, three methods that need to be called together), nor how each library compares to other libraries with similar features. Ultimately, developers need to select the most suitable library from the available ones for the task at hand.

When selecting a library to reuse from a vast ecosystem, it becomes essential for developers to know the features offered by each library. Antoniol and Guéhéneuc [2] define a feature as "a set of data structures (i.e., fields and classes) and operations (i.e., functions and methods) participating in the realisation of the functionality". Kanda et al. [3] consider a set of API calls with a corresponding name as a feature. We generalise their definition in this paper so that *features* comprise the API elements that realise them, as well as a textual description of the offered functionality.

The documentation provided by the maintainers of a library is not always complete. Moreover, there is no standard for documenting the features of a library. Even if such a standard can be agreed upon for an ecosystem, it will take a considerable effort before tools and indexing platforms will be able to peruse the newly-documented information. At the time of writing, the MVN repository and NPM host approximately 414K and 1.6M libraries, respectively. Although at these scales, there is a need for automated feature documentation techniques, they are still few and far between.

An early feature documentation technique [3] automatically recognises similar API calls across the source code of the clients of a library. Once recognised, these can be augmented manually with documentation in natural language. More recently, Shen et al. [4] proposed NLI2Code, an IDE plugin capable of inserting a code template corresponding to the API usage required for a given verb phrase such as "set colour for an Excel cell". NLI2Code can therefore be seen as a natural language interface for a library. The plugin relies on a *functional feature extractor* that extracts verb phrases from Stack Overflow (SO) posts, and a *code pattern miner* that first extracts the API elements mentioned in the same posts and then mines GitHub client projects for frequent usage patterns.

In this paper, we present an approach to extract the features of a library using the library's JAR file and the corresponding SO posts alone. The latter present a unique opportunity for API mining because of their focus related to a task (i.e., API usages can be posted directly) and the natural language description around code snippets. The approach first computes the public API of the libraries in the ecosystem, and uses a

[1] http://www.modulecounts.com/

[2] https://mvnrepository.com/open-source/collections

custom island parser to recognise the usage of this API in SO snippets. The natural language surrounding these snippets is collected too. A clustering algorithm then clusters related API usages together based on the extracted data. The most common natural language terms in a cluster will form its description.

The contributions of this paper are as follows:

- We present a fully-automated and data-driven approach to extracting the functional features of a library from its SO posts. The approach combines hierarchical clustering with dynamic tree cutting to produce high-quality clusters of the API usage related to a library feature, described by the most commonly co-occurring natural language terms.
- We empirically study and select the configuration of parameters that produces the most cohesive clusters and evaluate the accuracy of the best configuration against both cookbook-style tutorials and client projects of 7 libraries.
- We share our implementation, the carefully curated datasets used in the evaluation, and our results[3].

This work has two major implications. First, developers can take advantage of the tool instantiation to compute a set of uncovered features for a given library and explore them. This exploration can enhance the process of library selection which tends to be biased towards older libraries, and the process of library comparison by founding it on uncovered feature sets. Second, library maintainers can improve the documentation for and the features of their libraries. Awareness of the features offered by competing libraries could affect future releases of a library and ultimately improve the ecosystem's health.

II. Motivation

Support for comparing libraries: Developers rely on metrics such as the number of downloads, votes, stars on GitHub, open and closed issues, release frequency, etc., when comparing libraries to use [5]. Several ecosystem indices such as NPMCompare[4], NPMTrends[5] or NPMS[6] have therefore been proposed. These enable comparing libraries in terms of metrics and quality scores. Figure 1 depicts an example of such a comparison provided by NPMCompare. LibComp [6] is a more recent IDE plugin that enables comparing metrics for a system's library dependencies and exploring alternatives.

The focus on popularity-based metrics, however, tends to bias the selection process towards the more popular and older libraries [7]. Newly-introduced libraries with superior functionality or with an API that requires less boilerplate code to use might take a while to surface. None of the software ecosystems is supported by tools that enable comparing libraries based on the features offered, nor on the API usage that is required to use them.

[3]https://github.com/cvelazquezr/saner-2022/
[4]https://npmcompare.com/
[5]https://www.npmtrends.com/
[6]https://npms.io/

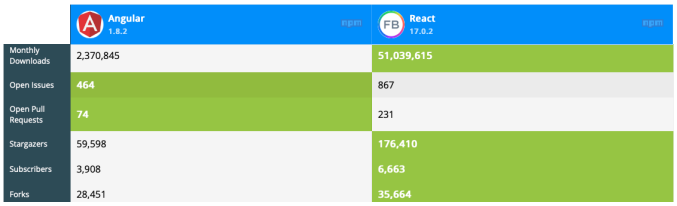

Fig. 1. Comparison of two libraries based on popularity metrics.

Support for exploring ecosystems: Tool support for the ecosystem exploration phase is lacking, with most official ecosystem indices being limited to browsing through community-curated or maintainer-provided categories and library tags. Some ecosystem indices support natural language queries against the short description of the indexed libraries, but such documentation might be lacking and there is no standard format for documenting the functional features provided by each library. With an automated means for uncovering library features, ecosystem indices could provide a richer browsing experience. For instance, users could inspect a list of features that are commonly or rarely implemented by the libraries in a selected category.

Ecosystem indices could also support queries against the uncovered textual descriptions, as well as the API elements that realise each feature. For instance, the query "resize an image", could return the API usage required for different libraries that support this task. Users could inspect the API usage required for a particular library and immediately navigate to a competing library for which less boilerplate code is required to use the same feature or of which the API is more aligned with the project's own coding conventions (e.g., a fluent or a regular API). Even more, developers that need multiple features, ideally want to inspect to what extent each library supports these features; e.g., developers might need to "resize an image" and "convert image to pdf". Current tool support does not allow developers to efficiently evaluate and compare candidate libraries w.r.t all the desired features.

III. Background

Hierarchical Clustering: Hierarchical clustering [8] is a technique that clusters elements based on the similarity of their attributes, and results in a hierarchy of clusters (often visualised as a dendrogram) so that elements in a child cluster also belong to the parent cluster. Intuitively, each layer in the hierarchy is characterised by its own maximum distance among the elements in its cluster. Bottom-up (or agglomerative) approaches initially form one cluster per element and merge these clusters repeatedly according to their similarity. The cluster at the root of the hierarchy encompasses all elements. Top-down (or divisive) approaches initially group all elements in a single cluster and repeatedly split until every element is on its own.

Static and Dynamic Tree Cutting: Hierarchical clustering results in a tree of clusters that needs to be cut at a given height cutoff in order to extract a unique clustering

of the dataset. In static cutting, each child below the given height cutoff forms a separate cluster. However, an appropriate height cutoff might be difficult to determine. A suboptimal clustering can result, especially for height cutoffs at which there are many similar elements and hence nested clusters. Alternatively, Langfelder et al. [9] propose to cut the tree branches dynamically based on its shape using either a top-down or a bottom-up algorithm. These have been shown to outperform static cutting in bioinformatics applications, and are amenable to complete automation.

IV. Approach

Figure 2 depicts an overview of our approach and the sequencing of its computational steps. The steps are denoted by a number (e.g., 1, 2), while letters indicate the data serving as input to or resulting from a step (e.g., A, B). Figures 3 and 5 zoom in on the internals of composite steps. The remainder of this section details each of the steps.

A. Data collection (Steps 1-3)

Our prototype implementation requires that the *groupID* and *artifactID* (e.g., `com.google.guava` and `guava` respectively) of at least one version of the library is published in the Maven Central repository. Once the user has selected a target library, the approach automatically collects information about its public API from its published JAR files (steps 2.1 and 2.2 in Figure 3) and example usages of this API from SO snippets (step 3 in Figure 2).

To collect information about the public API of a library (Figure 3), our approach considers all versions of the library published on Maven Central and thus downloads their JAR files (data 2.A). Next, step 2.2 extracts the names of the public classes in these JAR files using the *Apache BCEL*[7] library. *Apache BCEL* allows extracting information from Java Bytecode files (e.g., *.class) such as those in JAR artifacts. Although the public API of a library might evolve over time, we collect all class names that were once considered part of it. We opt for this strategy because API usage examples on SO rarely mention the exact library version that they exemplify. Moreover, the first automated approach [10] to determining the compatible library versions for the API usage within a given code snippet still produces version ranges.

Step 3 in Figure 2 collects SO answers from the SOTorrent dataset [11][8] that are likely to contain API usages of the selected library. Our approach uses a heuristic that considers all answers of which the question has been tagged with the name of the library (e.g., `guava`, `pdfbox`). More relaxed heuristics could also be used, such as requiring the library name to appear in the title or in the body of the question. We opted for the tags heuristic as it is computationally inexpensive and because its strictness minimizes false positives.

In addition to the body of the answer, step 3 also collects the title, tags, and the body of the question for further analysis.

B. Data processing (Steps 4-5)

Selecting SO answers based on the tags of their question ensures that the question concerns the target library. Unfortunately, not all answers associated with a question contain code snippets from which API usage examples can be extracted. For the Weka library[9], for instance, we found that around 57% of the answers about questions tagged with `weka` do not contain any code. Weka can be used as a library as well as an independent application for machine learning. Step 4 in Figure 2 therefore filters out all answers without code.

At this point in the approach, the class names within the public API of the selected library (data B) and the answers from SO with code snippets in which the API of the library is likely to be used (data C) have been collected. Step 5 identifies answers with code snippets that use the public API of the selected library. To this end, it relies on a robust parser generated by a custom-built island grammar. Parsers generated by an island grammar [12] focus on some constructs of interest (i.e., islands) and consider the remainder of the text to parse as irrelevant (i.e., water). They have been shown well-suited to parsing and lightweight analysis of code that is grammatically incomplete (e.g., a statement without a surrounding method) or that contains syntax errors (e.g., three dots instead of an expression) such as the snippets on SO. Ponzanelli et al. [13], [14], for instance, have used an island parser to recognize code snippets within the natural language of SO posts.

As our data processing focuses on the public API of a library, our own island parser focuses on the syntactic constructs in which method invocations can occur. Figure 4 depicts an example of its output for the code snippet shown on the left. For a single method invocation within a variable declaration or expression statement, it produces the name of the statically declared type of the receiver expression followed by the name of the invoked method. For a chain of successive method invocations, common for libraries with a fluent API, it produces the statically declared type of the first receiver expression followed by the names of the successively invoked methods. Note that it is not possible to resolve the receiver types of an invocation within the chain without resorting to more heavyweight program analyses.

We implemented our island parser, and its lightweight analysis that extracts API usage information using the *parboiled*[10] framework. The parser pushes constructs of interest to an internal stack whenever islands are encountered in the sea of water. For the encountered variable declarations and parameters, it pushes the variable type and identifier. For the encountered method invocations, it pushes the method name and the receiver expression (instance method) or type name (static method). Upon the completion of parsing, the lightweight analysis inspects the lexical scopes on the stack for potential matches between the identifiers in receiver expressions and those in variable and parameter declarations. The receiver expressions for which there is a match, are replaced by the type

[7]https://commons.apache.org/proper/commons-bcel/

[8]November 16th, 2020 version from https://zenodo.org/record/4287411

[9]https://www.cs.waikato.ac.nz/ml/weka/

[10]https://github.com/sirthias/parboiled/wiki

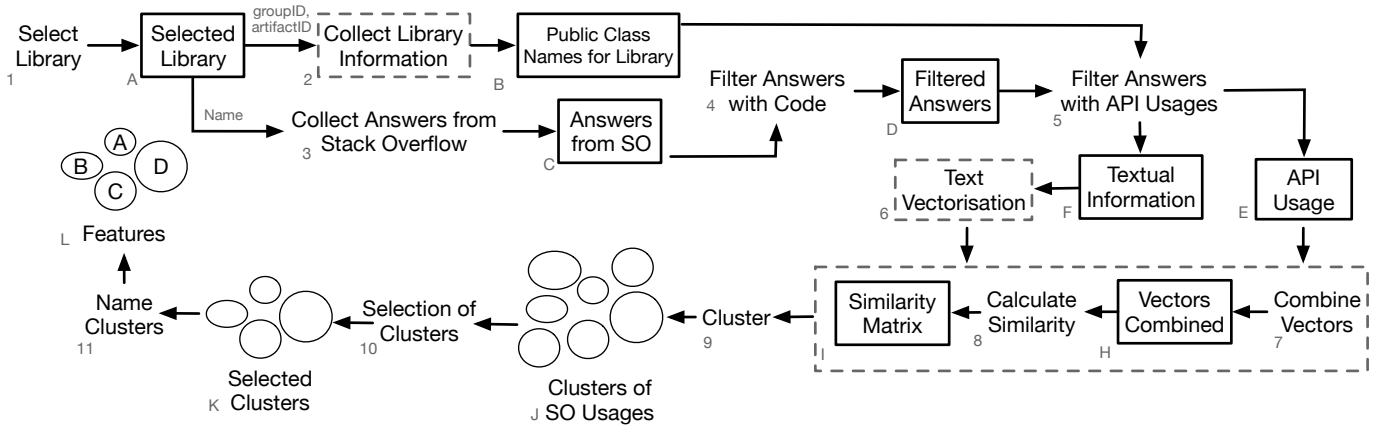

Fig. 2. Overview of the approach.

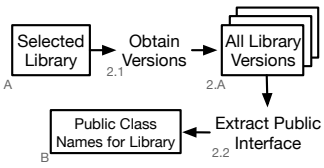

Fig. 3. Collecting the public interface of a library for all of its versions.

```
String html = "<p>An example link</p>";
Document doc = Jsoup.parse(html);
String text = doc.body().text();
```

Island Parser

```
Jsoup.parse
Document.body.text
```

Fig. 4. The API usages extracted by our island parser and its lightweight analysis for the code snippet on the left.

of their corresponding variable or parameter declaration. As a result, the analysis is able to retrieve the simple name of the receiver type of both instance and static method invocations.

With the API usage within each code snippet extracted by the island parser, step 5 merely has to match the type and method names against those of the public API of the library (i.e., data B). SO answers without any match are discarded. For the remaining answers, the extracted API usage information is kept (i.e., data E) along with the natural language in their body as well as the title and the natural language in the body of the question to which the answer belongs (i.e., data F).

C. Data transformation (Steps 6-8)

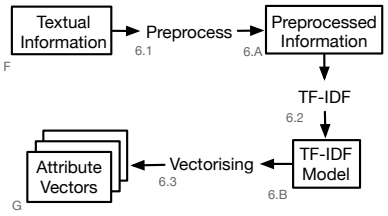

Fig. 5. Vectorisation of the textual information.

The next step in our pipeline is to transform the collected data. The transformation (i.e., box with dashed lines containing steps 7 and 8) independently process two inputs (e.g., text and code) and produces a similarity matrix per input.

The transformation process is twofold where 1) the text data is preprocessed and a TF-IDF model [15] is trained and 2) both code and text data are transformed into vectors. Figure 5 details how the vectorisation proceeds for the textual information (i.e., data F). Step 6.1 removes all stop words, the name of the programming language (i.e., Java), and the name of the library. Symbols such as commas, question marks, and dots are also removed in this preprocessing step.

Next, step 6.2 applies a TF-IDF vectoriser to the preprocessed data. Once its model has been trained (data 6.B), the vectors for 3 text-related attributes are calculated, namely: the body of the question, the body of the answer, and the title. Therefore, 3 text-based vectors will be produced for each of the considered SO answers (data G). In the case of code data (data E), no preprocessing or model training is needed. The following code-related attributes are sent to step 7: the original method names, the method names after splitting camel case, and the API usage itself as extracted by the island parser.

Step 7 combines a preconfigured selection of attributes. Text-based vectors are averaged and code-based attributes are concatenated. This is to support our empirical study into whether a single vector, a combination of some vectors, or all vectors produce more cohesive clusters. Regardless of the selected attribute to combine, the outcome of this step will be a single vector of numbers or code (e.g., classes and methods) per SO answer for text or code inputs, respectively. A similarity matrix resulting from computing a similarity metric between all vectors of all different answers, is finally calculated in step 8. We use cosine similarity to compute the distance between text-based vectors and Jaccard similarity for code-based vectors. Cosine similarity is a well-known metric to compute distances in text-related data, while Jaccard is more suitable for precise information such as code [16]. The generated matrices are the input to the next step.

D. Clustering, selecting and naming (Steps 9-11)

Step 9 applies hierarchical clustering to the similarity matrix of either API usages or the textual information extracted for each SO answer. The resulting clusters (i.e., data J) are groups of which the elements are close to one another yet far from the elements in other clusters. The ideal cluster would be that one in which the feature is clearly depicted. However, this might not always be the case. We use the local outlier factor (LOF) [17] (step 10) to check the most common elements within a cluster. In the cases where LOF does not determine a frequent element (or outlier), the cluster is discarded.

To facilitate interpreting the results, step 11 computes the most frequent terms that appear within each selected cluster. From the textual information (i.e., title, question and answer bodies) of each SO answer, we compute the semantic tree of each sentence [18]. For each noun and verb, we analyse the direct typed dependencies on another verb or noun to extract pairs of the form noun-verb or verb-noun.

Frequencies are calculated for each pair. Finally, the most frequent pairs are obtained through LOF as previously used for the most relevant code elements. Data L will ultimately contain a feature with related natural language terms (e.g., fig. 6) and API usages.

- use cache
- create cache
- store cache
- refresh cache
- be cache
- implement cache
- use cache builder
- want cache

Fig. 6. A name for an uncovered feature suggesting operations on cache.

V. Evaluation

Our approach is assessed by measuring the cohesiveness of the generated clusters, by comparing the obtained features with ground truth code in cookbooks or tutorials, and lastly by matching non-covered features in sampled GitHub projects. The evaluation of our approach focuses on the API calls of the generated features. The cluster names, i.e., the most frequent pair of terms that we extract as names of these clusters, need to be further researched and evaluated with controlled experiments (i.e., a user study) in follow-up work.

Specifically, our study aims to answer the following research questions:

- RQ1. Which combination of SO answer attributes produces the most cohesive clusters?
- RQ2. How similar are the automatically uncovered features to documented tutorial features?
- RQ3. To which extent do the uncovered features that do not match documented tutorial features correspond to actual API usage in client projects?

A. Selection of libraries

As candidate subjects for the evaluation of our approach, we consider any Java library from the Maven software ecosystem. We restrict candidate libraries to those that are used in SO snippets, so our approach can extract features from this usage. Moreover, candidate libraries need to have publicly available cookbooks or tutorials that illustrate the proper usage of the library. This is to enable comparing the features extracted by our approach with those documented for the library.

We selected the 50 most popular Maven libraries, measured in terms of GitHub repositories that use them according to the Libraries.io dataset[11]. From the initial population, we were able to find usage examples of features in tutorials or cookbooks in 7 libraries: Guava, HttpClient, JFreeChart, JSoup, PDFBox, Apache POI-OOXML and Quartz. These libraries form the subject systems for our evaluation, and cover diverse application domains.

Guava is a multi-purpose library collecting various auxiliaries related to data structures, I/O, caching, hashing, etc. HttpClient enables sending HTTP requests and process their responses. JFreeChart provides an API for creating and exporting graphics. JSoup facilitates manipulating elements present in web pages (e.g., links, divs). Similarly, Apache POI facilitates processing documents created with the Microsoft Office suite. We selected the sub-library dedicated to Excel worksheets (i.e., Apache POI-OOXML). Finally, Quartz allows launching and scheduling jobs programmatically.

Table I depicts the number of SO code snippets from where API usages are extracted for each library.

TABLE I
SO code snippets making use of the libraries.

	Guava	**HttpClient**	**JFreeChart**	**JSoup**	**PDFBox**	**POI**	**Quartz**
Snippets	1 522	628	882	3 465	945	2 010	511

B. Features Terminology

For disambiguation purposes, we consider it necessary to define some terminology before starting with the evaluation. We consider three types of constructs referred to as features or possibly containing a feature. First, we refer to the features generated by our approach as "uncovered features", i.e., clusters that contain related API usages. Second, we refer to the features which we extracted manually from cookbooks as "tutorial features". We use these tutorial features as a first ground truth to compare our uncovered features with. Third, the "GitHub API usages" correspond to usages of a library at the method level in a client project from GitHub that depends on the library. The GitHub API usages form our second ground truth in the evaluation below.

C. RQ1. Which combination of SO answer attributes produces the most cohesive clusters?

Design: This research question investigates which of the attributes related to API usage examples on SO produce more cohesive clusters and therefore more defined features (i.e., features with API calls present solely in one cluster).

To this end, we recall the attributes extracted from a SO post: the title of the question, the body of the question,

[11] https://libraries.io/api

the body of the answer, method names, method names split by CamelCase (CC), and complete API calls. Section IV categorised these attributes into a text attribute category and a code attribute category.

We check the cohesiveness of the clusters resulting from clustering solely on each attribute, as well as from clustering on combinations of attributes (e.g., question bodies combined with titles). Each vector in an attribute matrix represents the combination corresponding to one code snippet for that attribute (or the combination of several attributes). Once our approach obtains a matrix for each attribute, it proceeds to generate combinations with other matrices. The number of possible combinations equals 14 when considering combinations of size 1 to 3 for the attributes in each category; hence each category will produce 7 matrix combinations. For the 8 cases where a combination involves more than one attribute, a mean matrix (between the matrix attributes' vectors) results from the average of matrices in the combination. Therefore, 14 matrices, one per combination, will be analysed below.

We apply hierarchical clustering (see Section III) to each computed matrix and compute the quality of the resulting clusters. We determine the cohesiveness of the clustering of each combination using the Silhouette score [19]:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \tag{1}$$

where i represents one vector in the matrix, $a(i)$ calculates the distance between a vector i and its j cluster neighbours,

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i, j) \tag{2}$$

whereas $b(i)$ measures the distance between the vector i and other vectors j that are part of a different cluster,

$$b(i) = \min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i, j) \tag{3}$$

The Silhouette score is in the range [-1, 1]; values closer to 1 imply more cohesive clusters, while values closer to -1 imply clusters with incorrectly clustered elements. More cohesive clusters are clusters of which the elements are close to each other and at the same time further away from the elements of other clusters.

Results: Table II shows the Top-K Silhouette scores for all combinations with K ranging from 1 to 3. The best attribute combinations are very similar for most of the libraries. *Methods CC* appear in the Top-3 for all libraries. Other combinations are frequent throughout libraries such as *Methods* and *Methods + Methods CC*. *Methods*, *Methods CC* and *Methods + Methods CC* are the most frequent in the Top-1 with 3 appearances each (notice the same score for Top-2 and Top-3 in some cases).

The mean of the Top-1 scores is 0.48, indicating that clusters are sufficiently cohesive and differentiated. Their quality, on average, is on the borderline of what Kaufman and Rousseeuw [20] term "reasonable structure". A vast majority of the combinations are vectors formed by single attributes (e.g., *Methods*) denoting lower scores for combinations of more attributes. Finally, all combinations in Table II refer to code attributes instead of textual attributes. Code information is more concise and less ambiguous than textual information, as a feature can be expressed in many ways using natural language text. Moreover, the text information present in each SO answer is rather specific to the solved task and might not be related to other answers.

TABLE II
TOP-K SCORES FOR ALL COMBINATIONS OF LIBRARY ATTRIBUTES. BETWEEN PARENTHESES IS THE OBTAINED SCORE.

Library	K=1	K=2	K=3
Guava	Methods (0.6)	Methods CC (0.6)	Methods + Methods CC (0.59)
HttpClient	API Calls (0.46)	Methods (0.45)	Methods CC (0.45)
JFreeChart	Methods (0.38)	Methods CC (0.38)	Methods + Methods CC (0.38)
JSoup	API Calls (0.63)	Methods CC (0.51)	Methods (0.51)
PDFBox	Methods + Methods CC (0.37)	Methods CC (0.36)	Methods (0.36)
Apache-POI	Methods (0.52)	Methods CC (0.52)	Methods + Methods CC (0.51)
Quartz	Methods + Methods CC (0.45)	Methods CC (0.42)	API Calls (0.42)

Among the 14 considered attribute combinations, the singleton attribute combination *Methods CC* is the most frequent among those resulting in the Top-3 most cohesive clusterings. The Top-1 attribute combinations achieve a mean score of 0.48, and the formed clusters exhibit reasonable structure on average.

D. RQ2. How similar are the automatically uncovered features to documented tutorial features?

Design: For each of the 7 selected libraries, we manually extracted the usages from their tutorial features, similar to the outcome of the island parser in Figure 4.

Tutorial features are compared one by one with all uncovered clusters. A tutorial feature is matched to an uncovered feature based on the class and method names within each other. When matches occur, we store the uncovered feature identifier (i.e., a number) and the tutorial feature that was matched. Classes and methods in both types of features (i.e., tutorial and uncovered) could be fully or partially matched. For example, for *HttpClient* chained API calls are common in its SO answers. A tutorial may split these chains into separate invocations over several lines that produce the same outcome. Conversely, a single tutorial feature could illustrate the usage of more than one of our uncovered library features.

As API calls of a tutorial feature could be underrepresented in the uncovered feature's population, i.e., other classes or methods might appear in the latter which are not part of the former, we assess our approach using a relevance score:

$$relevance = avg(rel_c + rel_m) \quad (4)$$

where *relevance* is the average between rel_c and rel_m that measure the relevance of the uncovered feature classes and methods, respectively. In turn, rel_c and rel_m are defined as:

$$rel_c = \frac{\sum_{i=1}^{NCF} \frac{NC_i}{CA}}{NCF}, \; rel_m = \frac{\sum_{j=1}^{NMF} \frac{NM_j}{CA}}{NMF}$$

where NC_i is defined as the number of SO answers in the uncovered feature that contains the tutorial class i, while *CA* denotes the total number of SO answers in the uncovered feature. Similarly, NM_j denotes the number of SO answers in the uncovered feature that contains the tutorial method j. Finally, *NCF* and *NMF* represent the number of classes and methods respectively that comprise a tutorial feature.

rel_c measures the average percentage of SO answers in the uncovered feature that contain each tutorial feature class, and accordingly rel_m measures the percentage for methods. Both rel_c and rel_m are within the range [0, 1] thus bounding the relevance score in Equation (4) within the same range. Values closer to 0 or 1 indicate weak or strong relevance of the uncovered features w.r.t. the tutorial features, respectively. We aggregate relevance values for uncovered features of a library by averaging the relevance scores of each feature.

In a similar way to the relevance score, we compute an overflow metric that calculates the number of classes and methods, within an uncovered feature, that differ from the tutorial features. The overflow metric is defined as:

$$overflow = avg(over_c + over_m) \quad (5)$$

$$over_c = \frac{CF \backslash CC}{NCF} - \frac{CC \backslash CF}{NCC}, \; over_m = \frac{MF \backslash MC}{NMF} - \frac{MC \backslash MF}{NMC}$$

where *NCF* and *NMF* are defined in Equation (4), *NCC* and *NMC* are defined as the number of classes and methods within an uncovered feature, respectively. *CC* and *CF* are the classes of uncovered and tutorial features respectively; equivalently *MC* and *MF* represent the methods also in uncovered and tutorial features, respectively. The overflow ranges from $[-1; 1]$ with a score closer to -1 meaning an abundance of classes and methods in uncovered features not being present in tutorial features. In contrast, values closer to 1 allow concluding that the tutorials contain several elements from the uncovered features. An overflow metric close to 0 means that a tutorial and an uncovered feature are at an average distance of zero from each other. A value of 0 can arise in two possible scenarios: 1) there is a similar disparity between $over_c$ and $over_m$ but with different signs (e.g.,-0.5 and 0.5), or 2) there is a similar set of classes and methods between the tutorial and the uncovered feature. The last scenario (number 2) is desirable, i.e., the ground truth and uncovered features are closer to each other.

Finally, we measure the accuracy of the matches, i.e., how many uncovered features match tutorial ones. Other metrics considering the total number of features (e.g., recall) are not realistic since those found in tutorials might only represent a fragment of the features of a library.

Results: Table III presents the results for the matches between uncovered and tutorial features. The second column of the table shows the number of tutorial features for each of the libraries, while the third column displays the number of matches between the uncovered features and the tutorial features. Also, we report the number of tutorial features that are not found in our data (fourth column), and the scores of accuracy, relevance and overflow in the last four columns. Note that a tutorial feature might be present in several uncovered features with different relevance scores, however, we select the cluster with the highest relevance. A higher relevance implies similar classes and methods between the features in comparison. The averages of highest relevances and overflows per library are shown in Table III.

TABLE III
ANALYSIS OF THE MATCHED FEATURES PER LIBRARY.

Library	No. Feat.	Match.	Not Found	Acc.	R-Acc.	Relv.	Over.
Guava	30	22	7	0.73	0.97	0.71	-0.28
HttpClient	12	9	3	0.75	1.0	0.71	-0.23
JFreeChart	9	8	0	0.89	0.89	0.73	-0.26
JSoup	15	11	0	0.73	0.73	0.95	-0.51
PDFBox	15	8	4	0.53	0.73	0.82	-0.30
Apache-POI	20	10	3	0.50	0.59	0.75	-0.39
Quartz	22	12	5	0.55	0.71	0.65	-0.58

A total of 123 features (sum of *No. Feat.*) were collected. Our approach achieves high accuracy for the *JFreeChart* library with 89% and at least 50% for all libraries in the analysis. It achieves more modest accuracy scores on *Apache-POI*, *PDFBox* and *Quartz* with 50%, 53% and 55% respectively.

Interestingly, the features that our approach did not recover comprise API calls that were not found in SO. For *Guava*, 7 out of 30 tutorial features were missing from the SO data, representing around 23% of Guava's features. The same goes for *HttpClient*, for which 3 out of 12 (25%) of the tutorial features did not have SO snippets involving their API calls. Column *R-Acc.* therefore depicts the accuracy of uncovered features, but this time only considering the tutorial features of which the API usage also appears in SO snippets. Here, our approach improves its accuracy by 13% w.r.t. the *Acc.* column.

The relevance results (*Relv.* column in Table III) shows good performance generally. The average relevance achieved for the libraries is 76% with a standard deviation of ± 9, with the lowest score being achieved for *Quartz*. The highest relevance score, 95%, was achieved on the *JSoup* library. This library does not have a great variety of classes and methods, hence, the generated clusters cover the majority of classes and methods referenced in tutorials.

The overflow metric (*Over.* column in Table III) shows negative scores on average for all libraries indicating the presence of classes and methods in the uncovered features not being in the tutorial features. The overflow results combined with the relevance scores suggest that tutorial features are in their majority covered, but there might be additional classes or methods in the uncovered feature that are frequently used.

Uncovered features have an average accuracy of 67%, which increases to 80% when comparing only to tutorial features with calls that appear in SO code snippets. High relevance scores indicate that uncovered features are highly similar to tutorial features. Our uncovered features are very likely to contain additional classes or methods, as indicated by the scores on the overflow metric.

E. RQ3. To which extent do the uncovered features that do not match documented tutorial features correspond to actual API usage in client projects?

Design: Tutorial features might cover but a fragment of the features of a library. As the complete set of features is not well-defined by developers nor by the community, tutorials do not allow us to fully evaluate the features uncovered by our approach. We therefore look into alternative sources to verify the unmatched uncovered clusters from RQ2. We select GitHub as such an alternative source. Numerous GitHub projects can depend on a particular library, each using a diverse set of library features.

To obtain the API usage from GitHub clients for a particular library, we first query the API of Libraries.io to retrieve candidate client projects with a declared dependency on the library under analysis. Note that these client projects declare a dependency to libraries in their configuration file, but do not necessarily contain actual API usage of the library in their source code [21]. We therefore discard GitHub client projects without actual API usage of the library under analysis. We continue this filtering process until we have collected, for each library in our evaluation, a statistically significant sample of GitHub client projects from Libraries.io with a confidence level of 95% and a confidence interval of 5%[12].

We clone the sampled GitHub repositories, extract the bodies of their method declarations, and obtain the API usages within each method using our island parser. We only extract and keep the API usage that belongs to the library under analysis.

We then compare the resulting API usages to the non-matched clusters from RQ2. In this experimental setting, the relevance (Equation (4)) and overflow (Equation (5)) scores are also computed to compare GitHub API usages with the uncovered features produced by our approach.

TABLE IV
CLIENT PROJECTS INFORMATION FROM GITHUB.

Library	Clients	Sample	No. Methods
Guava	103 158	383	7 198
HttpClient	71 540	382	1 672
JFreeChart	2 830	338	3 829
JSoup	33 203	380	4 575
PDFBox	3 703	348	2 017
Apache-POI	35 517	380	1 769
Quartz	17 460	376	1 737

Results: Table IV presents information about the GitHub projects that use each library. The number of client projects (second column in Table IV) ranges from approximately 3K to more than 100K. This is expected since a multi-purpose library such as *Guava* is useful across application domains.

[12]https://www.surveysystem.com/sscalc.htm

TABLE V
NEWLY MATCHED FEATURES FROM GITHUB CLIENT PROJECTS.

Library	No. Feat.	M-RQ2	U-RQ2	M-RQ3	%	Relv.	Over.
Guava	110	22	14	91	95	0.45	0.00
HttpClient	38	9	7	24	77	0.50	0.05
JFreeChart	70	8	5	55	85	0.35	0.02
JSoup	81	11	11	52	74	0.60	-0.21
PDFBox	44	8	7	32	86	0.42	0.01
Apache-POI	81	10	9	53	74	0.50	-0.10
Quartz	31	12	5	19	73	0.48	0.00

Table V presents the information w.r.t. matching the uncovered features with GitHub API usages. Note that we split our results since we (a) perform a second evaluation for the uncovered features that were matched to tutorials in *RQ2* and (b) evaluate the remaining clusters, i.e., those features that were not matched against tutorials in *RQ2*. Columns *No. Feat.* and *M-RQ2* refer to the number of uncovered features extracted by our approach and the number of matched tutorial features from the cookbooks (cfr. RQ2) respectively. The features matched in RQ2 (column *M-RQ2*) were inspected to check unique matches in column *U-RQ2*.

Features in *U-RQ2* were removed from the uncovered features (column *No. Feat.*) to compute the new matches. Values in the column *M-RQ3* show the number of new matched features found in GitHub API usages and their coverage percentages (column %).

GitHub API usages cover the unmatched uncovered features to a high degree, with the highest coverage scores being realised for *Guava*, *PDFBox* and *JFreeChart* with 95%, 86%, and 85% respectively. This finding reveals that many tutorials paint an incomplete picture of a library's features. Moreover, some of the unmatched ones might still be used in a project outside of our sample or they might represent rare features.

We measure once again the relevance (column *Relv.*) of the uncovered features w.r.t. the GitHub usages to quantify the similarity of the two types of data. Relevance in Table V is on average lower in this case. Lower relevance is related to the overflow metrics (column *Over.*) which have positive values for most of the libraries —in contrast to Table III. Although overflow scores are positive, they are still close to zero; however, a closer inspection reveals that the overflow of classes remains mostly negative whereas methods are in their majority shifted to positive values, hence the mean seems steady towards zero. This finding (reflected in Table VI) suggests that the GitHub methods from which we extracted the API usage either tend to use several uncovered features together in their body, or that the way developers group API calls together into client methods does not align with the boundaries of our grouping into uncovered features.

TABLE VI
DETAILED OVERFLOW SCORES PER LIBRARY.

	Guava	HttpClient	JFreeChart	JSoup	PDFBox	POI	Quartz
Classes	-0.18	0.03	-0.21	-0.27	-0.15	-0.30	-0.24
Methods	0.18	0.06	0.26	-0.15	0.17	0.10	0.24

The majority of the uncovered features are found in sampled GitHub client projects. On average, the uncovered features are considered relevant with respect to the API usage within GitHub client methods. GitHub usages might be less focused or encompass more than one of our features.

VI. DISCUSSION

Clusters as features: Our approach uses a set of attributes about SO posts to obtain a coherent clustering of API usages. The evaluation shows that the best clusters are formed under attribute configurations that include the methods that are involved in an API usage (see Table II). The combination *Methods + Methods CC* might initially introduce a repetition of the information, especially in methods with no CamelCase style. However, many of the libraries (6 out of 7) in Table II present a different score for the mentioned combination and their constituents, except for *JFreeChart*. Although small, these differences indicate some degree of CamelCase usage and therefore, different information to be analysed.

Some of the resulting clusters even contain calls to API methods that belong to different classes. For example, one of the clusters obtained for *Guava* has the `transform` API method as the most frequent element. This feature is intended for transforming one collection data type into another. As a result, several classes implement it such as Lists and Maps.

Features can also include several API method calls that have to be used together. Listings 1 and 2 depicts examples from *JFreeChart* and *PDFBox* respectively.

```
1 // draw line, use chart, add jpanel, ...
2 XYSeries.add(...);
3 ChartFactory.createXYLineChart(...);
4 // show value, show percentage, change size, ...
5 DefaultPieDataset.setValue(...); // or
6 DefaultCategoryDataset.setValue(...)
7 ChartFactory.createPiechart(...);
```

Listing 1: Examples of features uncovered for JFreeChart.

```
1 // convert image, convert pdf
2 PDDocument.load(...);
3 PDDocument.getPage(...);
4 PDPageContentStream.drawImage(...);
5 PDDocument.save(...);
6 // merge file, reuse PDFMergerUtility, ...
7 PDFMergerUtility.mergeDocuments(...);
```

Listing 2: Examples of features uncovered for PDFBox.

However, our approach does also produce some clusters that are difficult to interpret as features of a library. Listing 3 depicts clusters that consist of a single API call, which should not be used on its own but together with other calls. Line 2 could be used as a closing statement for an HTTP response, line 4 as an additional attribute to the axis configuration of charts, and lines 6 and 8 as predicates within a conditional. Such clusters are likely to result from calls that appear in isolation without further context within the text of an answer.

```
1 // Feature from HttpClient
2 CloseableHttpResponse.close();
3 // Feature from JFreeChart
4 CategoryPlot.setAxisOffset(...);
5 // Feature from Apache-POI
6 POIXMLDocument.hasOOXMLHeader(...);
7 // Feature from Quartz
8 JobDetail.equals(...);
```

Listing 3: Examples of non-feature clusters uncovered.

Limitations: The lack of previous work on automatically uncovering features for libraries represents a hurdle to the evaluation of our approach. There is no ground truth of features for a library. The feature set uncovered by our approach may not be complete itself. The results in Table V already indicated that some API usage in GitHub client projects does not match any of the features uncovered from SO data. This might be due to our approach producing an incorrectly formed cluster that does not correspond to a feature, or due to the API usage in the client project corresponding to a rarely used feature for which little SO data is available.

As such cases were rare, however, the evaluation strengthens our believe that crowd knowledge (i.e., SO data) covers a large number of library features. The fact that the majority of uncovered features was found in client projects strengthens our confidence in their correctness.

Potential Impact: As revealed by our evaluation, a substantial number of library features is not documented in any tutorial for their library. Our approach can help users to understand the features offered by a library, and library maintainers to document these features. Once applied to all libraries within a software ecosystem, our approach will enable comparing competing libraries in terms of their feature sets. This might help developers in selecting the most appropriate library for the task at hand, and library maintainers to assess the ease of use of the APIs of different libraries for the same feature. This could stimulate cross-pollination between libraries and increase the dissemination of novel ideas within a library domain, thereby improving the health of the ecosystem.

VII. THREATS TO VALIDITY

Internal: We chose TF-IDF for the vectorisation of text attributes, even though more semantic approaches such as Word2Vec have recently been proposed. An initial experiment with Word2Vec resulted in less than five clusters per library where one cluster grouped most of the elements, but other vectorisation algorithms might still outperform TF-IDF. For the clustering algorithm, we selected hierarchical clustering based on its speed, effectiveness, and prior successes in API

classification [16]. Other clustering algorithms such as e.g., K-Means or DBSCAN might produce different results.

Construct: Our approach considers an SO post to be related to a library if the library's name appears among the tags of the post. This heuristic minimizes false positives, but more relaxed ones could be used (e.g., library name in question body). There is also a minor risk that some of the GitHub projects considered for RQ3 might have copied their code from SO. However, we believe that the impact of this threat is very low because the results of RQ2 align with the results of RQ1.

External: The success of our approach cannot be generalised readily to other Maven libraries, as it depends on the extent that their features are used in SO answers. Our approach might fail to uncover features for less popular libraries for which there is little usage in SO answers (see Table I).

Conclusion: Our selection of the best-performing attribute combinations relies on the Silhouette score. The score is a proven technique for analysing the quality of clusterings [19], [20], [22].

VIII. Related Work

Feature Uncovering: Kanda et al. [3] present an approach to uncovering functional features from the source code of an Android application. The approach exploits the assumption that apps with similar features make similar calls to the Android SDK. It is semi-automated as call sequences that are frequent across apps need to be described manually as a feature. Our approach differs in intent and nature, but shares the definition of a functional feature as a set of related API calls with a textual description.

Zhang and Hou [23] analyse forum discussions in search of problematic APIs. Their tool Haystack identifies sentences with a negative sentiment that contain API tokens. Through the Stanford NLP toolkit, it recommends API feature names (e.g., resize jscrollpane). We also use the Stanford toolkit to extract pairs of words as candidate names for the features. However, our features are accompanied by API code examples. Moreover, we focus on all API usages that our tool finds in SO and not only on posts with negative sentiments.

Guzman et al. [24] and Shah et al. [25] uncover features from app reviews using NLP techniques such as collocation finding and topic modelling. Al-Subaihin et al. [26] use developer-provided descriptions to this end, while Guo et al. [27] extract textual information from their GUI instead.

The NLI2Code plugin for IntelliJ proposed by Shen et al. [4] bridges the natural language and functional perspectives on features. The method invocations in the same posts are used to retrieve candidate API elements, of which API usages are extracted from client projects on GitHub. Finally, a frequent pattern mining algorithm is used to obtain the frequent API usage patterns. Our approach solely uses API usages from SO posts, and therefore relies on island parsing to extract API usage from incomplete and syntactically incorrect snippets.

API Usage Analysis: Amann et al. [28] survey several API usage miners targeting API misuse detection. They differ in the techniques used to extract and subsequently analyse API usage for frequent patterns. Our approach extracts API usage from SO snippets through an island parser, and subsequently clusters them according to attribute-based similarities.

Zhong and Mei [29] found that single-type usages (i.e., of a single API class) are as common as multi-type usages (i.e., involving more than one API class). API usages in code snippets follow such findings, forming in many cases clusters with a handful of related classes whereas in others the diversity of API classes is much higher. Gu et al. [30] propose to cluster API usage graphs by embedding them into a continuous space using a graph kernel and applying spectral clustering. Our approach uses an attribute vector combination upon which we perform hierarchical clustering with dynamic tree cutting.

Library Recommendation: The CLAN tool [31] computes the similarity of applications based on term-document matrices and Latent Semantic Indexing between a user query and the API semantics. Thung et al. [32] use association rule mining and collaborative filtering to propose libraries given other libraries in clients. Chen et al. [33] extracts analogical libraries based on SO tag sentences where libraries are used. A multi-objective search algorithm such as NSGA-II is used by Ouni et al. [34] to suggest the most appropriate library to a specific system. Req2Lib [35] uses instead a Sequence-to-Sequence Deep Learning architecture to propose the most suitable library given a description of requirements, addressing in this way the cold-start problem inherently adhered to recommendations. Our goal instead is to extract features that might be the foundation for library comparison and recommendation.

Library Documentation: Treude and Robillard [36] extract meaningful sentences from SO which contribute to API documentation. Tutorials are fragmented and then analysed by FRAPT [37], an unsupervised approach that automatically finds relevant textual fragments around code usages. Souza et al. [38] generate cookbooks from API usages based on SO *how-to* posts. Velázquez-Rodríguez and De Roover [39] propose an approach to recommend multiple tags for libraries based on their Java byte code. We do not focus on improving library documentation, although we foresee a potential use for our generated features in this area.

IX. Conclusion

This paper proposes a data-driven approach to uncovering library features from API usage in Stack Overflow answers. The approach combines hierarchical clustering with dynamic tree cutting to group the answers based on vectorised attributes.

We evaluated our approach on 7 popular libraries with cookbook-style documentation of their features. We achieve good performance for our uncovered features since accuracies and relevances are 67% and 76% on average, respectively. Uncovered features are highly covered by GitHub API usages (81% on average), however they might be part of larger usages as our results indicate.

X. Acknowledgements

This research was carried out in the context of the Excellence of Science project 30446992 *SECO-ASSIST* financed by FWO-Vlaanderen and F.R.S.-FNRS.

REFERENCES

[1] S. A. Ajila and D. Wu, "Empirical study of the effects of open source adoption on software development economics," *Journal of Systems and Software*, vol. 80, no. 9, pp. 1517–1529, 2007, evaluation and Assessment in Software Engineering. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121207000076

[2] G. Antoniol and Y.-G. Guéhéneuc, "Feature Identification: A Novel Approach and a Case Study," *The 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 357–366, 2005.

[3] T. Kanda, Y. Manabe, T. Ishio, M. Matsushita, and K. Inoue, "Semi-Automatically Extracting Features from Source Code of Android Applications," *IEICE Transactions on Information and Systems*, vol. E96.D, no. 12, pp. 2857–2859, 2013.

[4] Q. Shen, S. Wu, Y. Zou, Z. Zhu, and B. Xie, "From API to NLI: A new interface for library reuse," *Journal of Systems and Software*, vol. 169, p. 110728, 2020.

[5] E. L. Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting Third-Party Libraries: The Practitioners' Perspective," *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 245–256, 2020.

[6] R. El-Hajj and S. Nadi, "LibComp: An IntelliJ Plugin for Comparing Java Libraries," *The 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1591–1595, 2020.

[7] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the Diversity of Software Package Popularity Metrics: An Empirical Study of NPM," in *The 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 589–593.

[8] O. Maimon and L. Rokach, *Data Mining and Knowledge Discovery Handbook. 2nd Edition.* Springer, 2010.

[9] P. Langfelder, B. Zhang, and S. Horvath, "Defining clusters from a hierarchical cluster tree: the Dynamic Tree Cut package for R," *Bioinformatics*, vol. 24, no. 5, pp. 719–720, 2008.

[10] A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "Identifying Versions of Libraries used in Stack Overflow Code Snippets," in *The 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 341–345.

[11] S. Baltes, L. Dumani, C. Treude, and S. Diehl, "SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts," in *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018, pp. 319–330.

[12] L. Moonen, "Generating Robust Parsers using Island Grammars," *The 8th Working Conference on Reverse Engineering*, pp. 13–22, 2001.

[13] L. Ponzanelli, A. Mocci, and M. Lanza, "StORMeD: Stack Overflow Ready Made Data," in *The 12th International Working Conference on Mining Software Repositories (MSR15)*, 2015.

[14] L. Ponzanelli, "Holistic Recommender Systems for Software Engineering," Ph.D. dissertation, Università della Svizzera italiana, Mar. 2017.

[15] H. C. Wu, R. W. P. Luk, K. F. Wong, and K. L. Kwok, "Interpreting TF-IDF Term Weights as Making Relevance Decisions," *ACM Transactions on Information Systems (TOIS)*, vol. 26, no. 3, pp. 1–37, 2008.

[16] J. Härtel, H. Aksu, and R. Lämmel, "Classification of APIs by Hierarchical Clustering," in *The 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 233–23 310.

[17] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "LOF: Identifying Density-Based Local Outliers," in *The 2000 ACM SIGMOD International Conference on Management of Data*, 2000, pp. 93–104.

[18] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *The 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2014, pp. 55–60.

[19] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, 1987.

[20] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 1990, vol. 344.

[21] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the Maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–44, 2021.

[22] K. R. Shahapure and C. Nicholas, "Cluster Quality Analysis Using Silhouette Score," in *The 7th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2020, pp. 747–748.

[23] Y. Zhang and D. Hou, "Extracting Problematic API Features from Forum Discussions," in *The 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 142–151.

[24] E. Guzman and W. Maalej, "How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews," *The 22nd International Requirements Engineering Conference (RE)*, pp. 153–162, 2014.

[25] F. A. Shah, Y. Sabanin, and D. Pfahl, "Feature-Based Evaluation of Competing Apps," *The International Workshop on App Market Analytics*, pp. 15–21, 2016.

[26] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang, "Clustering Mobile Apps Based on Mined Textual Features," *The 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10, 2016.

[27] Y. Guo, Y. Li, Z. Yang, and X. Chen, "What's Inside My App?: Understanding Feature Redundancy in Mobile Apps," *The 26th Conference on Program Comprehension*, pp. 266–276, 2018.

[28] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A Systematic Evaluation of Static API-Misuse Detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2017.

[29] H. Zhong and H. Mei, "An Empirical Study on API Usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2016.

[30] X. Gu, H. Zhang, and S. Kim, "CodeKernel: A Graph Kernel based Approach to the Selection of API Usage Examples," *The 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, vol. 00, pp. 590–601, 2019.

[31] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting Similar Software Applications," *The 34th International Conference on Software Engineering (ICSE)*, vol. 1, pp. 364–374, 2012.

[32] F. Thung, D. Lo, and J. Lawall, "Automated Library Recommendation," *The 20th Working Conference on Reverse Engineering (WCRE)*, pp. 182–191, 2013.

[33] C. Chen, S. Gao, and Z. Xing, "Mining Analogical Libraries in Q&A Discussions," *The 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 338–348, 2016.

[34] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[35] Z. Sun, Y. Liu, Z. Cheng, C. Yang, and P. Che, "Req2Lib: A Semantic Neural Model for Software Library Recommendation," *The 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, vol. 00, pp. 542–546, 2020.

[36] C. Treude and M. P. Robillard, "Augmenting API Documentation with Insights from Stack Overflow," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 392–403.

[37] H. Jiang, J. Zhang, Z. Ren, and T. Zhang, "An Unsupervised Approach for Discovering Relevant Tutorial Fragments for APIs," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 38–48.

[38] L. B. Souza, E. C. Campos, F. Madeiral, K. Paixão, A. M. Rocha, and M. d. A. Maia, "Bootstrapping cookbooks for APIs from crowd knowledge on Stack Overflow," *Information and Software Technology*, vol. 111, pp. 37–49, 2019.

[39] C. Velázquez-Rodríguez and C. De Roover, "MUTAMA: An Automated Multi-label Tagging Approach for Software Libraries on Maven," in *The 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2020, pp. 254–258.