Summarizing Software API Usage Examples using Clustering Techniques

Nikolaos Katirtzis[1,2], Themistoklis Diamantopoulos[3], and Charles Sutton[2]

[1] Hotels.com, London, United Kingdom
[2] School of Informatics, University of Edinburgh, Edinburgh, United Kingdom
[3] Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Thessaloniki, Greece
`nkatirtzis@ed-alumni.net`, `thdiaman@issel.ee.auth.gr`, `csutton@ed.ac.uk`

Abstract. As developers often use third-party libraries to facilitate software development, the lack of proper API documentation for these libraries undermines their reuse potential. And although several approaches extract usage examples for libraries, they are usually tied to specific language implementations, while their produced examples are often redundant and are not presented as concise and readable snippets. In this work, we propose a novel approach that extracts API call sequences from client source code and clusters them to produce a diverse set of source code snippets that effectively covers the target API. We further construct a summarization algorithm to present concise and readable snippets to the users. Upon evaluating our system on software libraries, we indicate that it achieves high coverage in API methods, while the produced snippets are of high quality and closely match handwritten examples.

Keywords: API Usage Mining, Documentation, Source Code Reuse, Code Summarization, Mining Software Repositories

1 Introduction

Third-party libraries and frameworks are an integral part of current software systems. Access to the functionality of a library is typically offered by its API, which may consist of numerous classes and methods. However, as noted by multiple studies [24, 30], APIs often lack proper examples and documentation and, in general, sufficient explanation on how to be used. Thus, developers often use general-purpose or specialized code search engines (CSEs), and Question-Answering (QA) communities, such as Stack Overflow, in order to find possible API usages. However, the search process in these services can be time consuming [13], while the source code snippets provided in web sites and QA communities might be difficult to recognise, ambiguous, or incomplete [28, 29].

As a result, several researchers have studied the problem of API usage mining, which can be described as automatically identifying a set of patterns that characterize how an API is typically used from a corpus of client code [11]. There are two main types of API mining methods. First are methods that return API call sequences, using techniques such as frequent sequence mining [31–33], clustering [25, 31, 33], and probabilistic modeling [9]. Though interesting, API call sequences do not always describe important information like method arguments and control flow, and their output cannot be directly included in one's code.

A second class of approaches automatically produces source code snippets which, compared to API call sequences, provide more information to the developer, and are more similar to human-written examples. Methods for mining snippets, however, tend to rely on detailed semantic analysis, including program slicing [5,13–15] and symbolic execution [5], which can make them more difficult to deploy to new languages. Furthermore, certain approaches do not use any clustering techniques, thus resulting to a redundant and non-diverse set of API soure code snippets [20], which is not representative as it only uses a few API methods as noted by Fowkes and Sutton [9]. On the other hand, approaches that do use clustering techniques are usually limited to their choice of clustering algorithms [34] and/or use feature sets that are language-specific [13–15].

In this paper, we propose *CLAMS (Clustering for API Mining of Snippets)*, an approach for mining API usage examples that lies between snippet and sequence mining methods, which ensures lower complexity and thus could apply more readily to other languages. The basic idea is to cluster a large set of usage examples based on their API calls, generate summarized versions for the top snippets of each cluster, and then select the most representative snippet from each cluster, using a tree edit distance metric on the ASTs. This results in a diverse set of examples in the form of concise and readable source code snippets. Our method is entirely data-driven, requiring only syntactic information from the source code, and so could be easily applied to other programming languages. We evaluate CLAMS on a set of popular libraries, where we illustrate how its results are more diverse in terms of API methods than those of other approaches, and assess to what extent the snippets match human-written examples.

2 Related Work

Several studies have pointed out the importance of API documentation in the form of examples when investigating API usability [18, 22] and API adoption in cases of highly evolving APIs [16]. Different approaches have thus been presented to find or create such examples; from systems that search for examples on web pages [28], to ones that mine such examples from client code located in source code repositories [5], or even from video tutorials [23]. Mining examples from client source code has been a typical approach for Source Code-Based Recommendation Systems (*SCoReS*) [19]. Such methods are distinguished according to their output which can be either source code snippets or API call sequences.

2.1 Systems that Output API Call Sequences

One of the first systems to mine API usage patterns is *MAPO* [32] which employs *frequent sequence mining* [10] to identify common usage patterns. Although the latest version of the system outputs the API call sequences along with their associated snippets [33], it is still more of a sequence-based approach, as it presents the code of the client method without performing any summarization, while it also does not consider the structure of the source code snippets.

Wang et al. [31] argue that MAPO outputs a large number of usage patterns, many of which are redundant. The authors therefore define *scalability*, *succinctness* and *high-coverage* as the required characteristics of an API miner and construct UP-Miner, a system that mines probabilistic graphs of API method calls and extracts more useful patterns than MAPO. However, the presentation of such graphs can be overwhelming when compared to ranked lists.

Recently, Fowkes and Sutton [9] proposed a method for mining API usage patterns called PAM, which uses probabilistic machine learning to mine a less redundant and more representative set of patterns than MAPO or UP-Miner. This paper also introduced an automated evaluation framework, using handwritten library usage examples from Github, which we adapt in the present work.

2.2 Systems that Output Source Code Snippets

A typical snippet mining system is *eXoaDocs* [13–15] that employs slicing techniques to summarize snippets retrieved from online sources into useful documentation examples, which are further organized using clustering techniques. However, clustering is performed using semantic feature vectors approximated by the Deckard tool [12], and such features are not straightforward to get extracted for different programming languages. Furthermore, eXoaDocs only targets usage examples of single API methods, as its feature vectors do not include information for mining frequent patterns with multiple API method calls.

APIMiner [20] introduces a summarization algorithm that uses slicing to preserve only the API-relevant statements of the source code. Further work by the same authors [4] incorporates association rule techniques, and employs an improved version of the summarization algorithm, with the aim of resolving variable types and adding descriptive comments. Yet the system does not cluster similar examples, while most examples show the usage of a single API method.

Even when slicing is employed in the aforementioned systems, the examples often contain extraneous statements (i.e. statements that could be removed as they are not related to the API), as noted by Buse and Weimer [5]. Hence, the authors introduce a system that synthesizes representative and well-typed usage examples using path-sensitive data flow analysis, clustering, and pattern abstraction. The snippets are complete and abstract, including abstract naming and helpful code, such as `try/catch` statements. However, the sophistication of their program analysis makes the system more complex [31], and increases the required effort for applying it to new programming languages.

Allamanis and Sutton [1] present a system for mining syntactic idioms, which are syntactic patterns that recur frequently and are closely related to snippets, and thus many of their mined patterns are API snippets. That method is language agnostic, as it relies only on ASTs, but uses a sophisticated statistical method based on Bayesian probabilistic grammars, which limits its scalability.

Although the aforementioned approaches can be effective in certain scenarios, they also have several drawbacks. First, most systems output API call sequences or other representations (e.g. call graphs), which may not be as helpful as snippets, both in terms of understanding and from a reuse perspective (e.g. adapting an example to fit one's own code). Several of the systems that output snippets do not group them into clusters and thus they do not provide a diverse set of usage examples, and even when clustering is employed, the set of features may not allow extending the approaches in other programming languages. Finally, certain systems do not provide concise and readable snippets as their source code summarization capabilities are limited.

In this work, we present a novel API usage mining system, CLAMS, to overcome the above limitations. CLAMS employs clustering to group similar snippets and the output examples are subsequently improved using a summarization algorithm. The algorithm performs heuristic transformations, such as variable type resolution and replacement of literals, while it also removes non-API statements, in order to output concise and readable snippets. Finally, the snippets are ranked in descending order of support and given along with comprehensive comments.

3 Methodology

3.1 System Overview

The architecture of the system is shown in Figure 1. The input for each library is a set of *Client Files* and the API of the library. The *API Call Extractor* generates a list of API call sequences from each method. The *Clustering Preprocessor* computes a distance matrix of the sequences, which is used by the *Clustering Engine* to cluster them. After that, the top (most representative) sequences from each cluster are selected (*Clustering Postprocessor*). The source code and the ASTs (from the *AST Extractor*) of these top snippets are given to the *Snippet Generator* that generates a summarized snippet for each of them. Finally, the *Snippet Selector* selects a single snippet from each cluster, and the output is given by the *Ranker* that ranks the examples in descending order of support.

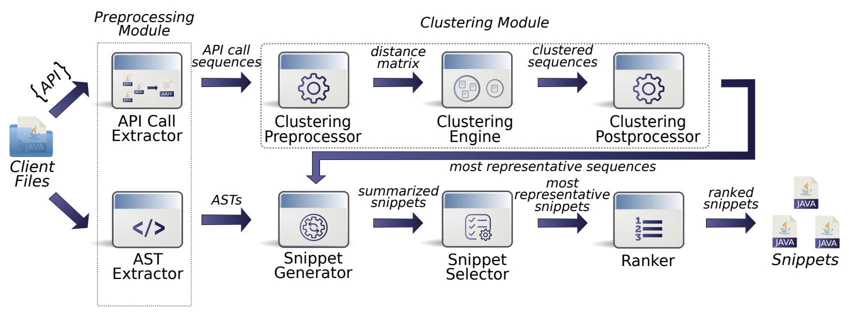


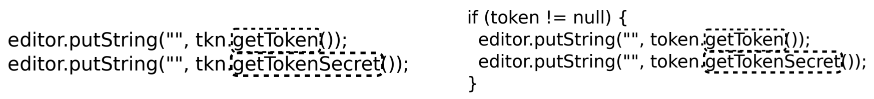
Fig. 1. Overview of the proposed system.

3.2 Preprocessing Module

The Preprocessing Module receives as input the client source code files and extracts their ASTs and their API call sequences. The *AST Extractor* employs srcML [8] to convert source code to an XML AST format, while the *API Call Extractor* extracts the API call sequences using the extractor provided by Fowkes and Sutton [9] which uses the Eclipse JDT parser to extract method calls using depth-first AST traversal.

3.3 Clustering Module

We perform clustering at sequence-level, instead of source code-level, this way considering all useful API information contained in the snippets. As an example, the snippets in Figures 2a and 2b, would be clustered together by our Clustering Engine as they contain the same API call sequence. Given the large number and the diversity of the files, our approach is more effective than a clustering that would consider the structure of the client code, while such a decision makes the deployment to new languages easier. Note however that we take into consideration the structure of clustered snippets at a later stage (see Section 3.5).



```
editor.putString("", tkn.getToken());
editor.putString("", tkn.getTokenSecret());
```

(a)

```
if (token != null) {
  editor.putString("", token.getToken());
  editor.putString("", token.getTokenSecret());
}
```

(b)

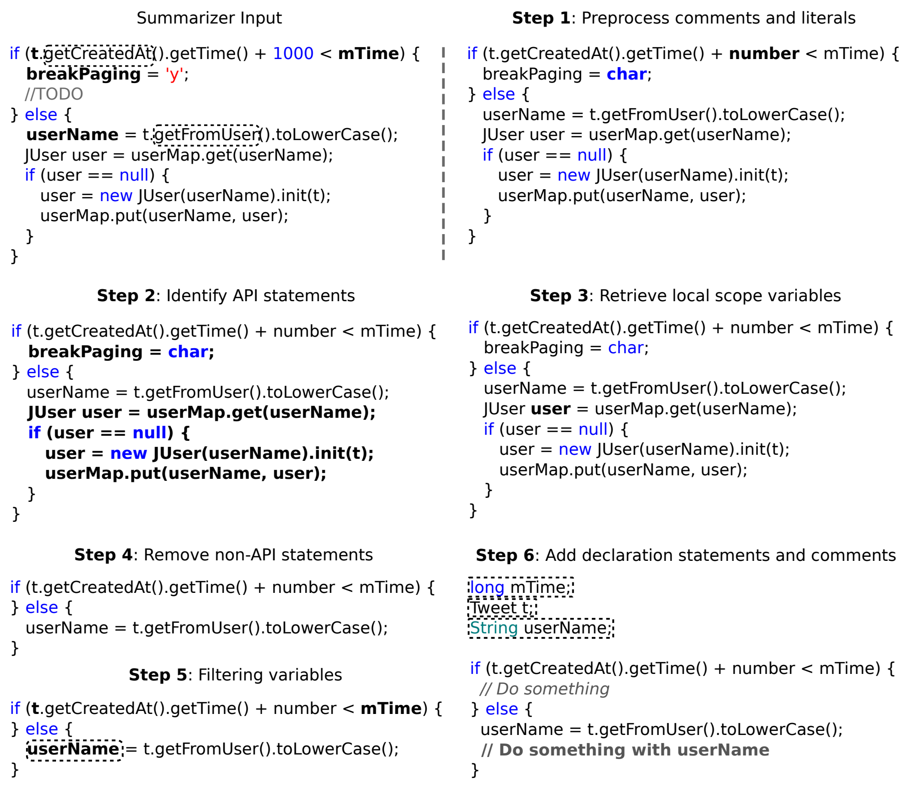
Fig. 2. The sample client code on the left side contains the same API calls with the client code on the right side, which are encircled in both snippets.

Our clustering methodology involves first generating a distance matrix and then clustering the sequences using this matrix. The *Clustering Preprocessor* uses the *Longest Common Subsequence (LCS)* between any two sequences in order to compute their distance and then create the distance matrix. Given two sequences S_1 and S_2, their LCS distance is defined as:

$$LCS\_dist\left(S_1, S_2\right) = 1 - 2 \cdot \frac{\left|LCS\left(S_1, S_2\right)\right|}{\left|S_1\right| + \left|S_2\right|} \tag{1}$$

where $|S_1|$ and $|S_2|$ are the lengths of S_1 and S_2, and $|LCS\left(S_1, S_2\right)|$ is the length of their LCS. Given the distance matrix, the *Clustering Engine* explores the k-medoids algorithm which is based on the implementation provided by Bauckhage [3], and the hierarchical version of DBSCAN, known as *HDBSCAN* [7], which makes use of the implementation provided by McInnes et al. [17].

The next step is to retrieve the source code associated with the most representative sequence of each cluster (*Clustering Postprocessor*). Given, however,



Summarizer Input

```
if (t.getCreatedAt().getTime() + 1000 < mTime) {
  breakPaging = 'y';
  //TODO
} else {
  userName = t.getFromUser().toLowerCase();
  JUser user = userMap.get(userName);
  if (user == null) {
    user = new JUser(userName).init(t);
    userMap.put(userName, user);
  }
}
```

Step 1: Preprocess comments and literals

```
if (t.getCreatedAt().getTime() + number < mTime) {
  breakPaging = char;
} else {
  userName = t.getFromUser().toLowerCase();
  JUser user = userMap.get(userName);
  if (user == null) {
    user = new JUser(userName).init(t);
    userMap.put(userName, user);
  }
}
```

Step 2: Identify API statements

```
if (t.getCreatedAt().getTime() + number < mTime) {
  breakPaging = char;
} else {
  userName = t.getFromUser().toLowerCase();
  JUser user = userMap.get(userName);
  if (user == null) {
    user = new JUser(userName).init(t);
    userMap.put(userName, user);
  }
}
```

Step 3: Retrieve local scope variables

```
if (t.getCreatedAt().getTime() + number < mTime) {
  breakPaging = char;
} else {
  userName = t.getFromUser().toLowerCase();
  JUser user = userMap.get(userName);
  if (user == null) {
    user = new JUser(userName).init(t);
    userMap.put(userName, user);
  }
}
```

Step 4: Remove non-API statements

```
if (t.getCreatedAt().getTime() + number < mTime) {
} else {
  userName = t.getFromUser().toLowerCase();
}
```

Step 5: Filtering variables

```
if (t.getCreatedAt().getTime() + number < mTime) {
} else {
  userName = t.getFromUser().toLowerCase();
}
```

Step 6: Add declaration statements and comments

```
long mTime;
Tweet t;
String userName;

if (t.getCreatedAt().getTime() + number < mTime) {
  // Do something
} else {
  userName = t.getFromUser().toLowerCase();
  // Do something with userName
}
```

Fig. 3. Example summarization of source code snippet.

that each cluster may contain several snippets that are identical with respect to their sequences, we select multiple snippets for each cluster, this way retaining source code structure information, which shall be useful for selecting a single snippet (see Section 3.5). Our analysis showed that selecting all possible snippets did not further improve the results, thus we select n snippets and set n to 5 for our experiments, as trying higher values would not affect the results.

3.4 Snippet Generator

The *Snippet Generator* generates a summarized version for the top snippets. Our summarization method, a static, flow-insensitive, intra-procedural slicing approach, is presented in Figure 3. The input (Figure 3, top left) is the snippet source code, the list of its invoked API calls and a set of variables defined in its outer scope (encircled and highlighted in bold respectively).

At first, any comments are removed and literals are replaced by their srcML type, i.e. string, char, number or boolean (*Step 1*). In *Step 2*, the algorithm creates two lists, one for API and one for non-API statements (highlighted in bold), based on whether an API method is invoked or not in each statement.

Any *control flow statements* that include API statements in their code block are also retained (e.g. the else statement in Figure 3). In *Step 3*, the algorithm creates a list with all the variables that reside in the local scope of the snippet (highlighted in bold). This is followed by the removal of all non-API statements (*Step 4*), by traversing the AST in reverse (bottom-up) order.

In *Step 5*, the list of declared variables is filtered, and only those used in the summarized tree are retained (highlighted in bold). Moreover, the algorithm creates a list with all the variables that are declared in API statements and used only in non-API statements (encircled). In *Step 6*, the algorithm adds declarations (encircled) for the variables retrieved in Step 5. Furthermore, descriptive comments of the form "Do something with variable" (highlighted in bold) are added for the variables that are declared in API statements and used in non-API statements (retrieved also in Step 5). Finally, the algorithm adds "Do something" comments in any empty blocks (highlighted in italics).

Finally, note that our approach is quite simpler than static, syntax preserving slicing. E.g., static slicing would not remove any of the statements inside the `else` block, as the call to the `getFromUser` API method is assigned to a variable (`userName`), which is then used in the assignment of `user`. Our approach, on the other hand, performs a single pass over the AST, thus ensuring lower complexity, which in its turn reduces the overall complexity of our system.

3.5 Snippet Selector

The next step is to select a single snippet for each cluster. Given that the selected snippet has to be the most representative of the cluster, we select the one that is most similar to the other top snippets. The score between any two snippets is defined as the tree edit distance between their ASTs, computed using the AP-TED algorithm [21]. Given this metric, we create a matrix for each cluster, which contains the distance between any two top snippets of the cluster. Finally, we select the snippet with the minimum sum of distances in each cluster's matrix.

3.6 Ranker

We rank the snippets according to the support of their API call sequences, as in [9]. In specific, if the API call sequence of a snippet is a subsequence of the sequence of a file in the repository, then we claim that the file supports the snippet. For example, the snippet with API call sequence [twitter4j.Status.getUser, twitter4j.Status.getText], is supported by a file with sequence [twitter4j.Paging.<init>, twitter4j.Status.getUser, twitter4j.Status.getId, twitter4j.Status.getText, twitter4j.Status.getUser]. In this way, we compute the support for each snippet and create a complete ordering. Upon ordering the snippets, the AStyle formatter [2] is also used to fix the indentation and spacing.

3.7 Deploying to New Languages

Our methodology can be easily applied on different programming languages. The Preprocessing Module and the Snippet Selector make use of the source code's

AST, which is straightforward to extract in different languages. The Clustering Module and the Ranker use API call sequences and not any semantic features that are language-specific, while our summarization algorithm relies on statements and their control flow, a fundamental concept of imperative languages. Thus, extending our methodology to additional programming languages requires only the extraction of the AST of the source code, which can be done using appropriate tools (e.g. srcML), and possibly a minor adjustment on our summarization algorithm to conform to the AST schema extracted from different tools.

4 Evaluation

4.1 Evaluation Framework

We evaluate CLAMS on the APIs (all public methods) of 6 popular Java libraries, which were selected as they are popular (based on their GitHub stars and forks), cover various domains, and have handwritten examples to compare our snippets with. The libraries are shown in Table 1, along with certain statistics concerning the lines of code of their examples' directories (Example LOC) and the lines of code considered from GitHub as using their API methods (Client LOC).

Table 1. Summary of the evaluation dataset.

Project	Package Name	Client LOC	Example LOC
Apache Camel	org.apache.camel	141,454	15,256
Drools	org.drools	187,809	15,390
Restlet Framework	org.restlet	208,395	41,078
Twitter4j	twitter4j	96,020	6,560
Project Wonder	com.webobjects	375,064	37,181
Apache Wicket	org.apache.wicket	564,418	33,025

To further strengthen our hypothesis, we also employ an automated method for evaluating our system, to allow quantitative comparison of its different variants. To assess whether the snippets of CLAMS are representative, we look for "gold standard" examples online, as writing our own examples would be time-consuming and lead to subjective results.

We focus our evaluation on the 4 research questions of Figure 4. RQ1 and RQ2 refer to summarization and clustering respectively and will be evaluated with respect to handwritten examples. For RQ3 we assess the API coverage achieved by CLAMS versus the ones achieved by the API mining systems MAPO [32, 33] and UP-Miner [31]. RQ4 will determine whether the extra information of source code snippets when compared to API call sequences is useful to developers.

We consider four configurations for our system: *NaiveNoSum*, *NaiveSum*, *KMedoidsSum*, and *HDBSCANSum*. To reveal the effect of clustering sequences, the first two configurations do not use any clustering and only group identical

RQ1: How much more concise, readable, and precise with respect to handwritten examples are the snippets after summarization?
RQ2: Do more powerful clustering techniques, that cluster similar rather than identical sequences, lead to snippets that more closely match handwritten examples?
RQ3: Does our tool mine more diverse patterns than other existing approaches?
RQ4: Do snippets match handwritten examples more than API call sequences?

Fig. 4. Research Questions (RQs) to be evaluated.

sequences together, while the last two use the k-medoids and the *HDBSCAN* algorithms, respectively. Also the first configuration (*NaiveNoSum*) does not employ our summarizer, while all others do, so that we can measure its effect.

We define metrics to assess the *readability*, *conciseness*, and *quality* of the returned snippets. For readability, we use the metric defined by Buse and Weimer [6] which is based on human studies and agrees with a large set of human annotators. Given a Java source code file, the tool provided by Buse and Weimer [27] outputs a value in the range [0.0, 1.0], where a higher value indicates a more readable snippet. For conciseness, we use the number of *Physical Lines of Code* (*PLOCs*). Both metrics have already been used for the evaluation of similar systems [5]. For quality, as a proxy measure we use the similarity of the set of returned snippets to a set of handwritten examples from the module's developers.

We define the similarity of a snippet s given a set of examples E as *snippet precision*. First, we define a set E_s with all the examples in E that have exactly the same API calls with snippet s. After that, we compute the similarity of s with all matching examples $e \in E_s$ by splitting the code into sets of tokens and applying set similarity metrics[4]. Tokenization is performed using a Java code tokenizer and the tokens are cleaned by removing symbols (e.g. brackets, etc.) and comments, and by replacing literals (i.e. numbers, etc.) with their respective types. The precision of s is the maximum of its similarities with all $e \in E_s$:

$$Prec(s) = max_{e \in E_s} \left\{ \frac{|T_s \cap T_e|}{|T_s|} \right\} \tag{2}$$

where T_s and T_e are the set of tokens of the snippet s and of the example e, respectively. Finally, if no example has exactly the same API calls as the snippet (i.e. $E_s = \varnothing$), then snippet precision is set to zero. Given the snippet precision, we also define the average snippet precision for n snippets $s_1, s_2, \ldots, s_n$ as:

$$AvgPrec(n) = \frac{1}{n} \sum_{i=1}^{n} Prec(s_i) \tag{3}$$

[4] Our decision to apply set similarity metrics instead of an edit distance metric is based on the fact that the latter one is heavily affected and can be easily skewed by the order of the statements in the source code (e.g. nested levels, etc.), while it would not provide a fair comparison between snippets and sequences.

Similarly, average snippet precision at top k can be defined as:

$$AvgPrec@k = \frac{1}{k}\sum_{j=1}^{k} Prec@j \text{ where } Prec@j = \frac{1}{j}\sum_{i=1}^{j} Prec(s_i) \tag{4}$$

This metric is useful for evaluating our system which outputs ordered results, as it allows us to illustrate and draw conclusions for precision at different levels.

We also define coverage at k as the number of unique API methods contained in the top k snippets. This metric has already been defined in a similar manner by Fowkes and Sutton [9], who claim that a list of patterns with identical methods would be redundant, non-diverse, and thus not representative of the target API.

Finally, we measure additional information provided in source code snippets when compared with API call sequences. For each snippet we extract its *snippet-tokens* T_s, as defined in (2), and its *sequence-tokens* $T_s{}'$, which are extracted by the underlying API call sequence of the snippet, where each token is the name of an API method. Based on these sets, we define the *additional info* metric as:

$$AdditInfo = \frac{1}{m}\sum_{i=1}^{m} \frac{\max_{e \in E_s}\{|T_{s_i} \cap T_e|\}}{\max_{e \in E_s}\{|T_{s_i}{}' \cap T_e|\}} \tag{5}$$

where m is the number of snippets that match to at least one example.

4.2 Evaluation Results

RQ1: How much more concise, readable, and precise with respect to handwritten examples are the snippets after summarization? We evaluate how much reduction in the size of the snippets is achieved by the summarization algorithm, and the effect of summarization on the precision with respect to handwritten examples. If snippets have high or higher precision after summarization, then this indicates that the tokens removed by summarization are ones that do not typically appear in handwritten examples, and thus are possibly less relevant. For this purpose, we use the first two versions of our system, namely the *NaiveSum* and the *NaiveNoSum* versions. Both of them use the naive clustering technique, where only identical sequences are clustered together. Figures 5a and 5b depict the average readability of the snippets mined for each library and the average PLOCs, respectively. The readability of the mined snippets is almost doubled when performing summarization, while the snippets generated by the *NaiveSum* version are clearly smaller than those mined by *NaiveNoSum*. In fact, the majority of the snippets of *NaiveSum* contain less than 10 PLOCs, owing mainly to the non-API statements removal of the algorithm. On average, the summarization algorithm leads to 40% fewer PLOCS. Thus, we may argue that the snippets provided by our summarizer are readable and concise.

Apart from readability and conciseness, which are both regarded as highly desirable features [26], we further assess whether the summarizer produces snippets that closely match handwritten examples. Therefore, we plot the snippet precision at top k, in Figure 6a. The plot indicates a downward trend in precision

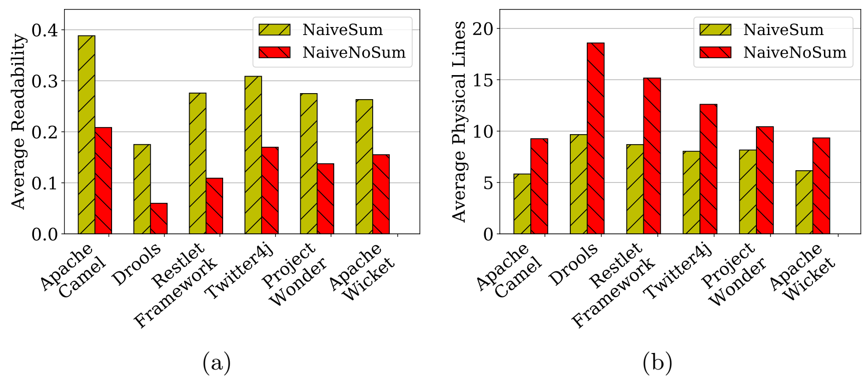


Fig. 5. Figures of (a) the average readability, and (b) the average PLOCs of the snippets, for each library, with (*NaiveSum*) and without (*NaiveNoSum*) summarization.

for both configurations, which is explained by the fact that the snippets of lower positions are more complex, as they normally contain a large number of API calls. In any case, it is clear that the version that uses the summarizer mines more precise snippets than the one not using it, for any value of k. E.g., for $k = 10$, the summarizer increases snippet precision from 0.27 to 0.35, indicating that no useful statements are removed and no irrelevant statements are added.

RQ2: Do more powerful clustering techniques, that cluster similar rather than identical sequences, lead to snippets that more closely match handwritten examples? In this experiment we compare *NaiveSum*, *KMedoidsSum*, and *HDBSCANSum* to assess the effect of applying different clustering techniques on the snippets. In order for the comparison to be fair, we use the same number of clusters for both k-medoids and HDBSCAN. Therefore, we first run HDBSCAN (setting its *min_cluster_size* parameter to 2), and then use the number of clusters generated by the algorithm for k-medoids. After that, we consider the top k results of the three versions, so that the comparison with the Naive method (that cannot be tuned) is also fair. Hence, we plot precision against coverage, in a similar manner to precision versus recall graphs. For this we use the snippet precision at k and coverage at k, while we make use of an *interpolated* version of the curve, where the precision value at each point is the maximum for the corresponding coverage value. Figure 6b depicts the curve for the top 100 snippets, where the areas under the curves are shaded. Area *A2* reveals the additional coverage in API methods achieved by *HDBSCANSum*, when compared to *NaiveSum* (*A1*), while *A3* shows the corresponding additional coverage of *KMedoidsSum*, when compared to *HDBSCANSum* (*A2*).

NaiveSum achieves slightly better precision than the versions using clustering, which is expected as most of its top snippets use the same API calls, and contain only a few API methods. As a consequence, however, its coverage is

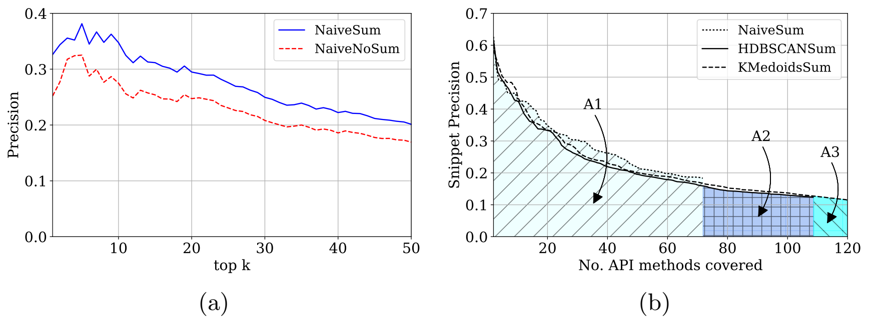


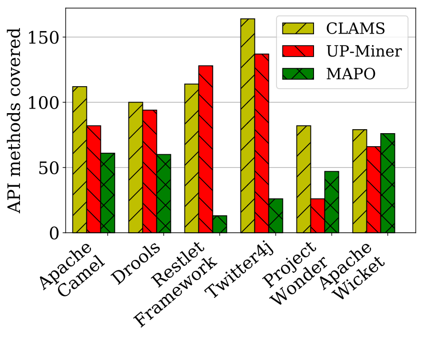
Fig. 6. Figures of (a) precision at top k, with (*NaiveSum*) or without (*NaiveNoSum*) summarization, and (b) the average interpolated snippet precision versus API coverage for three system versions (clustering algorithms), using the top 100 mined snippets.

quite low, due to the fact that only identical sequences are grouped together. Given that coverage is considered quite important when mining API usage examples [31], and that precision among all three configurations is similar, we may argue that *KMedoidsSum* and *HDBSCANSum* produce sufficiently precise and also more varying results for the developer. The differences between these two methods are mostly related to the separation among the clusters; the clusters created by *KMedoidsSum* are more separated and thus it achieves higher coverage, whereas *HDBSCANSum* has slightly higher precision. To achieve a trade-off between precision and coverage, we select *HDBSCANSum* for the last two RQs.

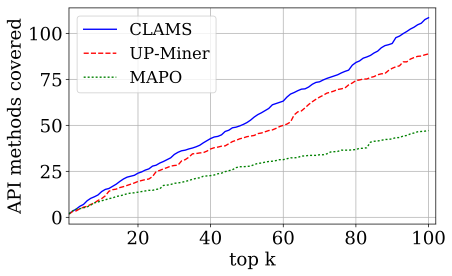
RQ3: Does our tool mine more diverse patterns than other existing approaches? For this research question, we evaluate the diversity of the examples of CLAMS to that of two API mining approaches, MAPO [32, 33] and UP-Miner [31], which were deemed most similar to our approach from a mining perspective (as it also works at sequence level)[5]. We measure diversity using the coverage at k. Figure 7a depicts the coverage in API methods for each approach and each library, while Figure 7b shows the average number of API methods covered at top k, using the top 100 examples of each approach.

The coverage by MAPO and UP-Miner is quite low, which is expected since both tools perform frequent sequence mining, thus generating several redundant patterns, a limitation noted also by Fowkes and Sutton [9]. On the other hand, our system integrates clustering techniques to reduce redundancy which is further eliminated by the fact that we select a single snippet from each cluster (Snippet Selector). Finally, the average coverage trend (Figure 7b) indicates that our tool mines more diverse sequences than the other two tools, regardless of the number of examples.

[5] Comparing with other tools was also hard, as most are unavailable, such as, e.g., the eXoaDocs web app (`http://exoa.postech.ac.kr/`) or the APIMiner website (`http://java.labsoft.dcc.ufmg.br/apimineride/resources/docs/reference/`).



(a)



(b)

Fig. 7. Graphs of the coverage in API methods achieved by CLAMS, MAPO, and UP-Miner, (a) for each project, and (b) on average, at top k, using the top 100 examples.

RQ4: Do source code snippets match handwritten examples more than API call sequences? Obviously source code snippets contain more tokens than API call sequences, but the additional tokens might not be useful. Therefore, we measure specifically whether the additional tokens that appear in snippets rather than sequences also appear in handwritten examples. Computing the average of the *additional info* metric for each library, we find that the average ratio between snippets-tokens and sequence-tokens, that are shared between snippets and corresponding examples, is 2.75. This means that presenting snippets instead of sequences leads to 2.75 times more information. By further plotting the additional information of the snippets for each library in Figure 8, we observe that snippets almost always provide at least twice as much valuable information. To further illustrate the contrast between snippets and sequences, we present an indicative snippet mined by CLAMS in Figure 9. Note, e.g., how the `try/catch` tokens are important, however not included in the sequence tokens.

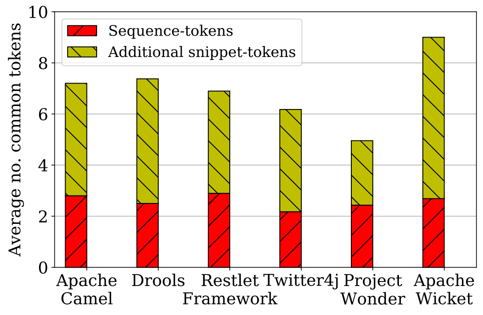


Fig. 8. Additional information revealed when mining snippets instead of sequences.

```
AccessToken accessToken;
String oauthToken;
String oAuthVerifier;
Twitter twitter;
try {
  accessToken = twitter.getOAuthAccessToken(oauthToken,oAuthVerifier);
  // Do something with accessToken
} catch (TwitterException e) {
  e.printStackTrace();
}
```

Fig. 9. Example snippet matched to handwritten example. Sequence-tokens are encircled and additional snippet-tokens are highlighted in bold.

Finally, we present the top 5 usage examples mined by CLAMS, MAPO and UP-Miner, in Figure 10. As one may observe, snippets provide useful information that is missing from sequences, including identifiers (e.g. `String secret`), control flow statements (e.g. `if-then-else` statements), etc. Moreover, snippets are easier to integrate into the source code of the developer, and thus facilitate reuse.

Interestingly, the snippet ranked second by CLAMS has not been matched to any handwritten example, although it has high support in the dataset. In fact, there is no example for the `setOauthConsumer` method of *Twitter4J*, which is one of its most popular methods. This illustrates how CLAMS can also extract snippets beyond those of the examples directory, which are valuable to developers.

```
Twitter mTwitter;
mTwitter = new TwitterFactory().getInstance();
// Do something with mTwitter

Twitter mTwitter;
final String CONSUMER_KEY;
final String CONSUMER_SECRET;
mTwitter = new TwitterFactory().getInstance();
mTwitter.setOAuthConsumer(CONSUMER_KEY,
      CONSUMER_SECRET);

BasicDBObject tweet;
Status status;
tweet.put(string, status.getUser().getScreenName());
tweet.put(string, status.getText());

String mConsumerKey;
Twitter mTwitter;
AccessToken mAccessToken;
String mSecretKey;
if (mAccessToken != null) {
  mTwitter.setOAuthConsumer(mConsumerKey, mSecretKey);
  mTwitter.setOAuthAccessToken(mAccessToken);
}

Twitter mTwitter;
String token;
String secret;
AccessToken at = new AccessToken(token, secret);
mTwitter.setOAuthAccessToken(at);
```

(a)

```
TwitterFactory.<init>
TwitterFactory.getInstance

Status.getUser
Status.getText

ConfigurationBuilder.<init>
ConfiguratiorBuilder.build

ConfigurationBuilder.<init>
TwitterFactory.<init>

ConfigurationBuilder.<init>
ConfigurationBuilder.setOAuthConsumerKey
```

(b)

```
TwitterFactory.getInstance
Twitter.setOAuthConsumer

TwitterFactory.<init>
TwitterFactory.getInstance
Twitter.setOAuthConsumer

Status.getUser
Status.getUser

ConfigurationBuilder.<init>
ConfigurationBuilder.build
TwitterFactory.<init>

ConfigurationBuilder.<init>
ConfigurationBuilder.build
TwitterFactory.<init>
TwitterFactory.getInstance
```

(c)

Fig. 10. Top 5 usage examples mined by (a) CLAMS, (b) MAPO, and (c) UP-Miner. The API methods for the examples of our system are highlighted.

5 Threats to Validity

The main threats to validity of our approach involve the choice of the evaluation metrics and the lack of comparison with snippet-based approaches. Concerning the metrics, snippet API coverage is typical when comparing API usage mining approaches. On the other hand, the choice of metrics for measuring snippet quality is indeed a subjective criterion. To address this threat, we have employed three metrics, for the conciseness (PLOCs), readability, and quality (similarity to real examples). Our evaluation indicates that CLAMS is effective on all of these axes. In addition, as these metrics are applied on snippets, computing them for sequence-based systems such as MAPO and UP-Miner was not possible. Finally, to evaluate whether CLAMS can be practically useful when developing software, we plan to conduct a developer survey. To this end, we have already performed a preliminary study on a team of 5 Java developers of Hotels.com, the results of which were encouraging. More details about the study can be found at `https://mast-group.github.io/clams/user-survey/` (omitted here due to space limitations).

Concerning the comparison with current approaches, we chose to compare CLAMS against sequence-based approaches (MAPO and UP-Miner), as the mining methodology is actually performed at sequence level. Nevertheless, comparing with snippet-based approaches would also be useful, not only as a proof of concept but also because it would allow us to comparatively evaluate CLAMS with regard to the snippet quality metrics mentioned in the previous paragraph. However, such a comparison was troublesome, as most current tools (including e.g., eXoaDocs, APIMiner, etc.) are currently unavailable (see RQ3 of Section 4.2). We may however note this comparison as an important point for future work, while we also choose to upload our code and findings online (`https://mast-group.github.io/clams/`) to facilitate future researchers that may face similar challenges.

6 Conclusion

In this paper we have proposed a novel approach for mining API usage examples in the form of source code snippets, from client code. Our system uses clustering techniques, as well as a summarization algorithm to mine useful, concise, and readable snippets. Our evaluation shows that snippet clustering leads to better precision versus coverage rate, while the summarization algorithm effectively increases the readability and decreases the size of the snippets. Finally, our tool offers diverse snippets that match handwritten examples better than sequences.

In future work, we plan to extend the approach used to retrieve the top mined sequences from each cluster. We could use a two-stage clustering approach where, after clustering the API call sequences, we could further cluster the snippets of the formed clusters, using a tree edit distance metric. This would allow retrieving snippets that use the same API call sequence, but differ in their structure.

References

1. Allamanis, M., Sutton, C.: Mining Idioms from Source Code. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 472–483. FSE 2014, ACM, New York, NY, USA (2014)
2. Artistic Style 3.0, avail. online: `http://astyle.sourceforge.net/`, [retrieved January, 2018]
3. Bauckhage, C.: Numpy/scipy Recipes for Data Science: k-Medoids Clustering. Tech. rep., University of Bonn (2015)
4. Borges, H.S., Valente, M.T.: Mining usage patterns for the Android API. PeerJ Computer Science 1, e12 (2015)
5. Buse, R.P.L., Weimer, W.: Synthesizing API Usage Examples. In: Proceedings of the 34th International Conference on Software Engineering. pp. 782–792. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012)
6. Buse, R.P., Weimer, W.R.: A Metric for Software Readability. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. pp. 121–130. ISSTA '08, ACM, New York, NY, USA (2008)
7. Campello, R.J.G.B., Moulavi, D., Sander, J.: Density-Based Clustering Based on Hierarchical Density Estimates, pp. 160–172. Springer, Berlin, Heidelberg (2013)
8. Collard, M.L., Decker, M.J., Maletic, J.I.: srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance. pp. 516–519. ICSM '13, IEEE Computer Society, Washington, DC, USA (2013)
9. Fowkes, J., Sutton, C.: Parameter-free Probabilistic API Mining Across GitHub. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 254–265. FSE 2016, ACM, New York, NY, USA (2016)
10. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques, pp. 1–38. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edn. (2011)
11. Ishag, M.I.M., Park, H.W., Li, D., Ryu, K.H.: Highlighting Current Issues in API Usage Mining to Enhance Software Reusability. In: Proceedings of the 15th International Conference on Software Engineering, Parallel and Distributed Systems. pp. 200–205. SEPADS '16, WSEAS (2016)
12. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In: Proceedings of the 29th International Conference on Software Engineering. pp. 96–105. ICSE '07, IEEE Computer Society, Washington, DC, USA (2007)
13. Kim, J., Lee, S., Hwang, S.w., Kim, S.: Adding Examples into Java Documents. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. pp. 540–544. ASE '09, IEEE, Washington, DC, USA (2009)
14. Kim, J., Lee, S., Hwang, S.w., Kim, S.: Towards an Intelligent Code Search Engine. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. pp. 1358–1363. AAAI'10, AAAI Press (2010)
15. Kim, J., Lee, S., Hwang, S.W., Kim, S.: Enriching Documents with Examples: A Corpus Mining Approach. ACM Trans. Inf. Syst. 31(1), 1:1–1:27 (2013)
16. McDonnell, T., Ray, B., Kim, M.: An Empirical Study of API Stability and Adoption in the Android Ecosystem. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance. pp. 70–79. ICSM '13, IEEE Computer Society, Washington, DC, USA (2013)
17. McInnes, L., Healy, J., Astels, S.: hdbscan: Hierarchical density based clustering. The Journal of Open Source Software 2(11) (2017)

18. McLellan, S.G., Roesler, A.W., Tempest, J.T., Spinuzzi, C.I.: Building more usable APIs. IEEE Software 15(3), 78–86 (1998)
19. Mens, K., Lozano, A.: Source Code-Based Recommendation Systems. In: Recommendation Systems in Software Engineering, pp. 93–130. Springer (2014)
20. Montandon, J.E., Borges, H., Felix, D., Valente, M.T.: Documenting APIs with examples: Lessons learned with the APIMiner platform. In: Proceedings of the 20th Working Conference on Reverse Engineering. pp. 401–408. WCRE '13 (2013)
21. Pawlik, M., Augsten, N.: Tree edit distance: Robust and memory-efficient. Information Systems 56(C), 157–173 (2016)
22. Piccioni, M., Furia, C.A., Meyer, B.: An Empirical Study of API Usability. In: Proceedings of the 7th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 5–14. ESEM '13 (2013)
23. Ponzanelli, L., Bavota, G., Mocci, A., Penta, M.D., Oliveto, R., Russo, B., Haiduc, S., Lanza, M.: CodeTube: Extracting Relevant Fragments from Software Development Video Tutorials. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 645–648. ICSE-C '16 (2016)
24. Robillard, M.P.: What Makes APIs Hard to Learn? Answers from Developers. IEEE Software 26(6), 27–34 (2009)
25. Saied, M.A., Benomar, O., Abdeen, H., Sahraoui, H.: Mining Multi-level API Usage Patterns. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). pp. 23–32 (2015)
26. Sillito, J., Maurer, F., Nasehi, S.M., Burns, C.: What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance. pp. 25–34. ICSM '12, IEEE Computer Society, Washington, DC, USA (2012)
27. Source Code Readability Metric, avail. online: `http://www.arrestedcomputing.com/readability`, [retrieved January, 2018]
28. Stylos, J., Faulring, A., Yang, Z., Myers, B.A.: Improving API Documentation Using API Usage Information. In: Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 119–126. VLHCC '09 (2009)
29. Subramanian, S., Inozemtseva, L., Holmes, R.: Live API Documentation. In: Proceedings of the 36th International Conference on Software Engineering. pp. 643–652. ICSE '14, ACM, New York, NY, USA (2014)
30. Uddin, G., Robillard, M.P.: How API Documentation Fails. IEEE Software 32(4), 68–75 (2015)
31. Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D.: Mining succinct and high-coverage api usage patterns from source code. In: Proceedings of the 10th Working Conference on Mining Software Repositories. pp. 319–328. MSR '13, IEEE Press, Piscataway, NJ, USA (2013)
32. Xie, T., Pei, J.: MAPO: Mining API Usages from Open Source Repositories. In: Proceedings of the 2006 International Workshop on Mining Software Repositories. pp. 54–57. MSR '06, ACM, New York, NY, USA (2006)
33. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: Mining and Recommending API Usage Patterns. In: Proceedings of the 23rd European Conference on Object-Oriented Programming. pp. 318–343. ECOOP '09, Springer-Verlag, Berlin, Heidelberg (2009)
34. Zhu, Z., Zou, Y., Xie, B., Jin, Y., Lin, Z., Zhang, L.: Mining api usage examples from test code. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 301–310. ICSME '14, IEEE Computer Society, Washington, DC, USA (2014)