# Detecting and Ranking API Usage Pattern in Large Source Code Repository: A LFM Based Approach

Jitong Zhao and Yan Liu(✉)

School of Software Engineering, Tongji University,
Cao'an Road 4800, Jiading District, Shanghai, China
{1410787,yanliu.sse}@tongji.edu.cn

**Abstract.** Code examples are key resources for helping programmers to learn correct Application Programming Interface (API) usages efficiently. However, most framework and library APIs fail in providing sufficient and adequate code examples in corresponding official documentations. Thus, it takes great programmers' efforts to browse and extract API usage examples from websites. To reduce such effort, this paper proposes a graph-based pattern-oriented mining approach, LFM-OUPD (Local fitness measure for detecting overlapping usage patterns) for API usage facility, that recommends proper API code examples from data analytics. API method queries are accepted from programmers and corresponding code files are collected from related API dataset. The detailed structural links among API method elements in conceptual source codes are captured and generate a code graph structure. Lancichinetti et al. proposed an overlapping community detecting algorithm (Local fitness measure, LFM), based on the local optimization of a fitness function. In LFM-OUPD, a mining algorithm based on LFM is presented to explore the division of method sequences in the directed source code element graph and detect candidates of different API usage patterns. Then a ranking approach is applied to obtain appropriate API usage pattern and code example candidates. A case study on Google Guava is conducted to evaluate the effectiveness of this approach.

**Keywords:** Graph mining · Source code mining · API usage recommendation · Data analytics

## 1 Introduction

Open source libraries and frameworks has mushroomed in the nearly ten years. Modern software industry cumulatively depends on third-party APIs provided by open source organizations [22]. Inevitably, programmers extensively leverage Application Programming Interfaces (APIs) to implement functionality and perform various tasks, which support code reuse and help unify programming


A. Holzinger et al. (Eds.): CD-MAKE 2017, LNCS 10410, pp. 41–56, 2017.
DOI: 10.1007/978-3-319-66808-6_4

experience. However, it is a tough task to select and utilize an appropriate API, as APIs have grown exponentially and become more inseparable from software development current days. For instance, to build a primary website using Spring-MVC framework, programmers need to invoke dozens of APIs, including data storage API, Java Messaging Service (JMS), ServletAPI and so on. The scope of all these APIs' skill sets are so broad that even the most experienced programmers need to spend lots of time to gain a thorough understanding of each API. Furthermore, various barriers factors [6,13] cause APIs hard to learn, such as insufficient or inadequate examples, unspecified issues with API's structural design, uncompleted or ambiguous documentation. Therefore, it is a critical job for assisting programmers to fully comprehend APIs effectively and efficiently with less effort.

Figure 1 shows the current state of practice: if programmers have questions about an API, they have to browse the documentation or code examples even articles on Internet for solutions manually. Also, they try to use APIs or ask colleagues sometimes.

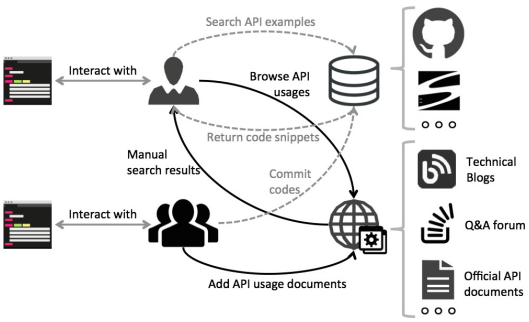


**Fig. 1.** Current state of practice

In these cases, source code examples often emerge as a acknowledged resource for programmers while learning usages of APIs [13]. This is also evident by the rapid development of code search engines (CSEs). While the results returned by CSEs are usually too massive for programmers to follow, so more researches have been done on various clustering and ranking algorithms in the stage of API usage results filter. Recent work [3,11,12,15] has helped programmers alleviate burden and better understand individual APIs. These approaches focus on GitHub source code analyzing, API usage mining, API pattern clustering and code candidates ranking.

API code usage information scatters on Internet including GitHub, official documents, and Stack overflow, however there is not a systematic knowledge base for this. These information usually suffers from deficiencies of irregular structure, which is similar to a social network. In this situation network community discovery methods are suitable to explore the programming information

raw data. In this paper, we propose an approach LFM-OUPD (Local fitness measure for detecting overlapping usage patterns) for API usage facility from data analytics perspective to help recommend proper code examples. We extract corresponding API method sequences from dataset, and construct links among API method elements, storing as graph structure. A mining algorithm based on Local fitness measure (LFM) is provided to detect usage patterns. After identifying and ranking API usage pattern with various features, we select the most appropriate code snippet for the programmers. At last, we conduct a comparison experiment between LFM, LFM-OUPD and MAPO [22], using a Google Guava[1] API case study. The results demonstrate that LFM-OUPD is more effective and adopted to actual APIs compared with the other two approaches.

The remainder of this paper is organized as follows. Section 2 summarizes related work including various recommending systems and technologies. Section 3 proposes an approach, LFM-OUPD, to detect and rank API usage pattern in software repository and recommend proper code examples for programmers. A case study on Google Guava is conducted to evaluate the effectiveness of this approach in Sect. 4. Lastly, we conclude this paper in Sect. 5.

## 2 Related Work

Helping programmers to learn how to use an API method has gained a considerable attention in recent year research. Several recommendation systems [3,11,12,15] have been designed to suggest relevant API usage examples for supporting programming tasks. They can be organized in the categories according to the data-mining mechanisms of their proposed techniques.

Some researches try to extract related information from numerous web pages, then deliver uniform views to programmers. APIExamples [18] performs in-depth analysis on the collected code snippets and descriptive texts from web pages, including usage examples clustering and ranking. It provides two kinds of user interaction style: an IDE plug-in and a web search portal.

Other contributions tried to leverage modern Q&A websites to provide efficient recommendations. Various technology forums offer concise answers and rich technical contexts including executable code snippets. Example Overflow [21] uses built-in social mechanisms of the popular technical forum, Stack Overflow[2] to recommend embeddable and high quality code.

The most related contributions are those interested in mining API usage pattern from common online code search engine and code snippet recommender. Current CSEs nearly wholly leverage text-oriented information-retrieval(IR) techniques that disregards inherent structure of source code. To address this problem, researchers have presented numerous approaches [9,14,15,17,19] to improve CSEs, further more, new recommendation engines are proposed. These are generally designed to work in the form of integrated development environment (IDE) plug-in to interact with programmers. MAPO [22] mines API usage

[1] https://github.com/google/guava.
[2] http://stackoverflow.com/.

pattern from large number of code snippets gathered by online code search engines such as Google Code Search.[3] In particular, based on the BIDE algorithm, MAPO combines frequent subsequence mining with clustering to mine closed sequential pattern. In addition, it provides a recommender integrated with Eclipse IDE.

The main limitations of all the contribution mentioned above are that they can't detect common API usage patterns, and do not performs in-depth analysis on the code graph structure. Our approach tries to resolves these limitations and presents a new dimension of the API usage pattern detection. First, we collect multiple API clients programs of the target studied API which help construct a graph structure. Then, a graph-based approach is presented to identify candidates of different API method sequences and rank them with three appropriateness metrics. Finally, our approach detects common API usage patterns which are used frequently in the same way by client programs and recommends them to the programmers.

## 3 LFM-OUPD: An Approach for Proper API Usage Pattern Recommendation

Aiming at addressing the issues during finding proper API code examples, we propose a recommendation approach, LFM-OUPD (Local fitness measure for detecting overlapping usage patterns). In LFM-OUPD, implicit API usage dependencies is modeled by API method call sequences, which are extracted from the API implementation code. As its name implies, API method call sequence is a series of API methods, which are used together to realize a specific function. LFM-OUPD takes these API usage dependencies as the basis of recommendation, which can make the suggested API usage patterns (Sect. 3.1) more appropriate for developers.

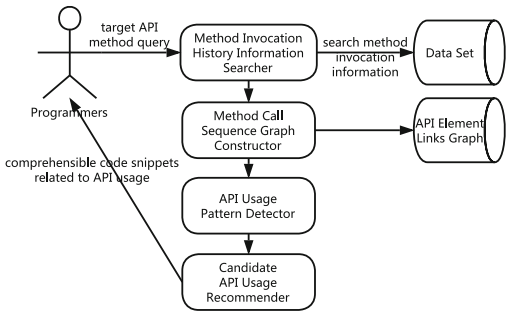


**Fig. 2.** Overview of LFM-OUPD

[3] http://www.google.com/codesearch.

Figure 2 shows the overview of LFM-OUPD. At beginning, an API programmer specifies the name of an API method that needs code examples as a query. Then, the API method invocation history information of target API method is collected from the existing dataset. LFM-OUPD utilizes a method call sequence graph constructor (Sect. 3.2) to capture the detailed structural links among API method elements in conceptual source codes and generate a code graph structure. An API usage pattern detector (Sect. 3.3) is proposed for classifying and mining different API usage patterns from the method call sequence graph. An optimized candidate API usage recommender (Sect. 3.4) is also required in processing and generating API usage pattern ranking list quickly. Finally, the most comprehension-friendly code snippets, which are related to the optimized API usage, are selected and recommended to the API programmers (Sect. 3.5).

### 3.1 API Usage Pattern

Before interpreting approach LFM-OUPD, we provide a detailed definition of API usage pattern. Our approach defines an API Usage Pattern (UP) as a sequence of API method calls. These API method calls have co-appeared relations, namely, they are always located closely in the client programs. Every API usage pattern is an exclusive subset of API call methods.

A fixed sequence of API method calls are usually located in different code snippets, which have the same co-appeared relations and are expected to be concluded into an API usage pattern. However, it is impossible to detect all the possible target method usage scenarios. An approach is required to identify possible API Usage Patterns from large amount of API's client programs and detect API's usage scenarios. Therefore, our approach is designed to capture out co-appeared relations among API's methods, and offer proper code examples relevant to the target API Usage Pattern.

### 3.2 Method Call Sequence Graph Constructor

**Preparing Dataset.** For most APIs with rich set of client code on Internet, potential sources files with code examples can be directly collected from existing data source. Client code hosted on GitHub can provide enough valuable scenarios on how to use various API methods. We need to collect code usage information from GitHub at project level for detecting APIs across projects and extract the references between methods of the client programs and the public methods of APIs based on Eclipse's JDT compiler [1]. The API usage dataset published by Sawant and Bacchelli [16] just meets our basic requirement. This dataset contains information about 5 APIs and how their public methods are utilized over the course of their entire lifetime by 20,263 projects. Thus, we utilized it in our approach to analyse the relevant API method usage information among code snippets.

**Generating API Method Call Sequences.** This section aims at generating API method call sequences of target query. In our approach, we consider the code locations as API method calls when there is a super constructor call, a method call or a class instance creation. To keep the issues simple, this paper does not consider method overloading situation in user code, and more research will be done in further.

Accepting the API method query from programmers, our approach performs search on API usage dataset to obtain target API method invocation history information. We locate all the class files where target API method ever appeared and collect these method invocation information in the class file level. We generate a method call sequence for each class file, which performs a sequential traversal to collect API method calls. For example, the corresponding method call sequence of statement 'static Cache<Integer, String> cache = CacheBuilder.newBuilder().expireAfterWrite(5, TimeUnit.SECONDS).build();' is as follows.

```
com.google.common.cache.CacheBuilder.newBuilder
com.google.common.cache.CacheBuilder.expireAfterWrite
com.google.common.cache.CacheBuilder.build
```

In addition, the information about class name, project name, url access of Github and so on are also collected and stored.

**Constructing Links Among API Method Elements.** After generating corresponding API method call sequences, the detailed structural links among API method elements in conceptual source codes are captured and generate a code graph structure illustrated in Fig. 3.

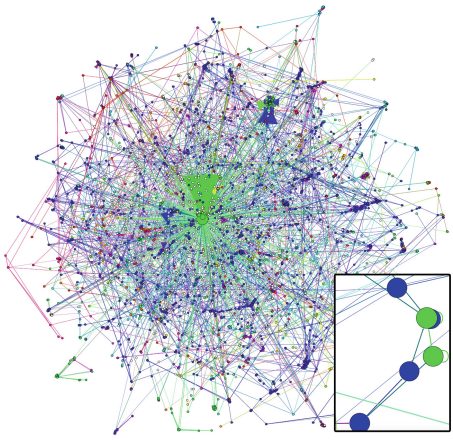

**Fig. 3.** Method sequence call graph

Intuitively, the method call sequences across different java class files compose a method call sequence graph together, with the linear method call sequence in

one file regarded as a sub-graph. In addition, the edge between two method nodes in this graph is weighed by the frequency of their co-appearance.

Nodes in the graph represent API methods, e.g., 'com.google.common.cache.CacheBuilder.newBuilder()'. The edge in this directed graph represents sequential flows from method to method based on the collected API method call sequences. Take for example, if there are three methods in a statement 'method1(); method2(); method3();', we construct a graph structure as 'method1() –> method2() –> method3()', which means method2() appears followed by method1() and method3() appears followed by method2() in this statement. In this case, method1() –> method3() is not allowed, which should strictly follow the appearance location sequence.

Also, the weight of the edge represents the time count that two methods are used together. The weight of edge is an important metric to evaluate the tightness of two API method, which contributes to detect API usage pattern in the latter section. Figure 4 shows an example of method call sequence graph. Method 'com.google.common.cache.CacheBuilder.maximumSize()' is located followed by method 'com.google.common.cache.CacheBuilder.newBuilder()' in java programs and the weight of their edge is 20. This means method newBuilder() and maximumSize() are used together and located closely in 20 java class files.

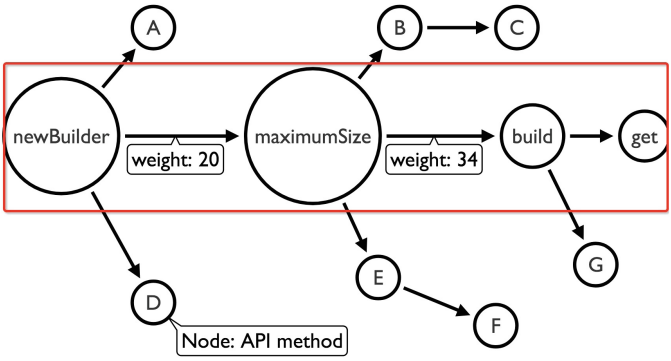


**Fig. 4.** Method sequence call graph demo

### 3.3 API Usage Pattern Detector

The method call sequence graph can be treated as a directed weighted complex network. We transfer the API usage pattern detecting question to the exploration question of high weight and high density method call sequences. Recent years, researchers propose many classical community detecting algorithms [5,8,10,20] to investigate complex networks and their natural characteristics. Lancichinetti et al. [7] proposed an overlapping community detecting algorithm (Local fitness measure, LFM), based on the local optimization of a fitness function. In this section, we propose a local extension algorithm of detecting overlapping usage patterns (LFM-OUPD) based on LFM.

**Directed Degree Centrality of Method Node.** In the directed weighted code graph, the large directed degree centrality of a method node usually means that the node is in the center of overall code network, and this node is also densely connected to the neighbouring method nodes. This kind of node is called core seed node of the directed source code graph. The directed degree centrality of one node is affected by weighted in-degree and out-degree of the node. We introduce a definition of directed degree centrality $C_i$ of node $i$ as follows,

$$C_i = \omega C_i^{in} + (1 - \omega) C_i^{out} \tag{1}$$

where $C_i^{in}$ represents the weighted in-degree of $node_j$, $C_i^{out}$ represents the weighted out-degree, and $\omega$ is the weight parameter, which is used to adjust the weight of in-degree and out-degree in the directed degree centrality.

**Neighbourhood Calculation Rules.** LFM-OUPD utilizes the idea of detecting communities through a local optimization of some metric, which has already been applied earlier [2,4]. This approach selects a core seed node as the initial sub-graph, and continuously extends this sub-graph by adding neighbour node until the fitness function of the usage pattern community reaches the maximization. Thus, It is helpful to introduce the neighbourhood calculation rules of a sub-graph community.

Assuming $UP_i$ is the $i$th element in a set of UPs, the neighbour nodes of $UP_i$ is identified as successor of leaf nodes in $UP_i$, and predecessor of root nodes in $UP_i$. Considering the characteristics of method sequence diagram, we don't take successor or predecessor of other nodes in $UP_i$ into consideration when finding neighbour nodes. As the code snippets are naturally existing as a single directed method chain diagram, the mined results are expected to be a method sequence diagram, which only one-way direction flow is permitted. It is noteworthy that, if there is a sub-ring in the method sequence call graph, we treat the sub-ring as a whole to determine whether it is a root or leaf node, and find their common successor or predecessor.

Figure 5 interprets the neighbourhood calculation rules. For a Usage Pattern with three API methods B, C and D, its leaf node is method D, and its root node is method B, so its neighbourhood are method A1, A2, E1 and E2. For a Usage Pattern with three API methods B, C and F, there is a sub-ring in it, that is method C and D. Here we treat method C and F as a whole leaf method node, so its neighbourhood are method A1, A2 and G.

**Directed Fitness Function of API Usage Patterns.** Directed Fitness Function of API Usage Patterns f($UP_i$) is utilized to measure the extent of impact when a node joins the API usage pattern $UP_i$. The basic assumption behind our algorithm is that the internal nodes in an API usage pattern is much more densely connected to each other compared with the relationships of other nodes. Thus, when f($UP_i$) reaches the peak, optimal division of a method sequence $UP_i$ is produced. If one node has a significant impact on its neighbourhood sequences, the node is divided into multiple API usage patterns, and then the overlapping

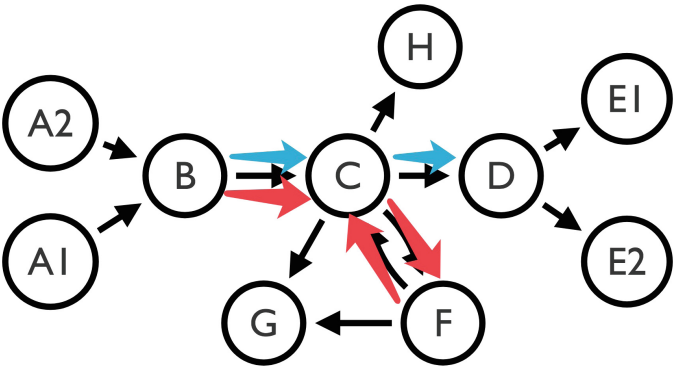


**Fig. 5.** Neighbourhood calculation example

method sequence structure is formed. Consider the characteristics of directed weighted edge graph structure, f($UP_i$) is defined as follows,

$$f(PU_i) = \frac{W_{all}^{in}}{(W_{all}^{in} + W_{leaf}^{in\_out} + W_{root}^{out\_in})\alpha} \tag{2}$$

where $W_{all}^{in}$ represents total weight of edges between nodes in $UP_i$, $W_{leaf}^{in\_out}$ represents total weight of edges directed from leaf nodes of $UP_i$ to neighbour nodes, $W_{root}^{out\_in}$ represents total weight of edges directed from neighbour nodes to root node of $UP_i$, and $\alpha$ is the threshold parameter used to control the size of the API usage pattern produced. Considering the characteristics of method call sequence graph and neighbour nodes calculation rules, the method sequence list is required to be single directed. Thus, the fitness function only cares about the edges directed to the root node of $UP_i$ from external nodes, and the edges directed from leaf nodes of $UP_i$ to the external nodes, when calculating weights between edges of $UP_i$ and outside.

**API Usage Patterns Detecting Algorithm.** Based on LFM, the core idea of our algorithm is to start from the core seed nodes in the API method call sequence graph, and constantly merge the neighbour to produce the API usage pattern. Firstly, the node with the largest directed degree centrality is found as the core seed node, and then the neighbour nodes with the greatest influence on the directed fitness function f($UP_i$) are merged continuously. Finally, the API usage pattern $UP_i$ is produced until the merging process comes to convergence. After forming a usage pattern, select the core node in the rest nodes to continue the next division of the usage pattern, and ultimately all the nodes in the graph are partitioned into usage patterns. Algorithm 1 describe the overall process of detecting API usage patterns.

**Algorithm 1.** API usage patterns detecting algorithm based on LFM

**Input:** Method sequence call graph $G(V, E, M)$, $\alpha$=0.3;
**Output:** Usage patterns set $UP$;
1: **for all** Method node $v_i \in V$ **do**
2: Calculate the directed degree centrality $C_i$ of $v_i$;
3: **end for**
4: Initialize the Usage patterns set $UP = \emptyset$;
5: Initialize the current usage pattern $UP_j = \emptyset$, $j = 0$;
6: Initialize the current method node set $V_c = V$;
7: Initialize the maximum fitness function $f_{max} = 0$;
8: **for** $V_c \neq \emptyset$ **do**
9: Set the seed node $v_s = \{v_i \mid \max_{v_i \in V_c} C_i\}$
10: Set $UP_j = v_s$;
11: Calculate the set of all neighbor nodes of $UP_j$, denoted by $V_n$;
12: **for all** $v_i \in V_n$ **do**
13: Add $v_i$ to $UP_j$ and calculate $f(UP_j)$;
14: Set $l = 0$;
15: **if** $f(UP_j) \geq f_{max}$ **then**
16: $f_{max} = f(UP_j)$, $l = i$;
17: **end if**
18: Delete $v_i$ from $UP_j$;
19: **end for**
20: **if** l==0 **then**
21: Add $UP_j$ to $UP$, $j = j + 1$;
22: **else**
23: Add $v_l$ to $UP_j$;
24: Do step 11-25;
25: **end if**
26: Update the current method node set $V_c = V_c - UP_j$
27: **end for**
28: **return** $UP$

### 3.4 Candidate API Usage Recommender

After usage patterns are detected by our approach, we select UP results containing user query(the target API method chose by the user) as API usage pattern candidates. Candidate API patterns ranking mechanism is also required to evaluate the score of usage pattern candidates and assist to pick appropriate ones. There are three appropriateness metrics, cohesiveness, availability and representation to calculate the final score, which is positive correlated with the ranking of an API usage pattern. The range of these three metrics is all between 0 and 1. The final score is the sum of these three metrics. The higher this score values, the more appropriate the API usage is for recommendation. After usage pattern candidate lists are recommended according to the ranking phase, users can select an interested usage pattern to browse related code examples.

**Cohesiveness.** The cohesiveness metric evaluates the aggregation of API method calls within a usage pattern. A usage pattern with higher cohesiveness is tended to be more frequent and significant. The cohesiveness metric for a usage pattern UP is determined by the weight of the usage pattern $W^{in}_{all}(UP)$ and the total weight of the method call sequence graph $TotalWeight(G)$. For example, if there is a usage pattern, whose $W^{in}_{all}(UP)$ is 49, and the $TotalWeight(G)$ of the method call sequence graph is 200, the $cohesiveness(UP)$ is 0.245. The formulation for cohesiveness is shown as follows.

$$cohesiveness(UP) = \frac{W^{in}_{all}(UP)}{TotalWeight(G)} \tag{3}$$

**Availability.** The availability metric evaluates the significance of a usage pattern. A usage pattern with higher availability is tended to be more universal for demonstrating a specific API method usage scenario. The availability metric for a usage pattern UP is determined by $CSNum(UP)$ and $CSNum(TargetQuery)$. $CSNum(UP)$ represents the number of code snippet (CS) where all the API method calls of the API usage pattern are used in. $CSNum(TargetQuery)$ represents the number of code snippet where the target method query call is ever used in. The formulation for availability is shown as follows.

$$availability(UP) = \frac{CSNum(UP)}{CSNum(TargetQuery)} \tag{4}$$

For example, the target method call is 'method1()' and there are 40 code snippets including 'method1()'. The detected usage pattern is the method call sequence 'method1, method2, method3()', and there are 15 code snippets containing these three methods. Then the *availability* is 15/40, that is 0.375.

**Representation.** The representation metric evaluates the portion of third-party method calls in the usage pattern calls. Methods invoked by a developer's own class will decrease the representation of a usage pattern. A usage pattern with higher representation is tended to be more representative and comprehensible. For a usage pattern with higher representation, the developer can filter useful information out of the noise during code review with less effort. The representation metric for a usage pattern UP is determined by the number of third-party method calls $ThirdPartyMCNum(UP)$ and the total number of method calls $MCNum(UP)$ appearing in the UP.

$$representation(UP) = \frac{ThirdPartyMCNum(UP)}{MCNum(UP)} \tag{5}$$

### 3.5 Candidate Code Examples Recommender

We use the following recommendation mechanism to rank code examples of target API usage pattern and pick the code snippets with the best demonstrative

effect: we prefer code snippet with less lines; prefer examples containing more comments; prefer code examples using less API methods apart from those in the target API usage pattern.

For a target API usage pattern, code snippets containing all the corresponding methods are gathered from the existing dataset. To select appropriate code snippet, we use three criterion, number of code lines $LineNum(CS)$, descriptive textural comments $CommentNum(CS)$ and number of API method calls of usage pattern $MCNum(UP)$ to calculate the ranking score. The formula is shown as follows.

$$Score(CS) = \frac{MCNum(UP)}{LineNum(CS)} + \frac{MCNum(UP)}{MCNum(CS)} + \frac{1}{CommentNum(CS)} \quad (6)$$

In addition, if there is not a comment in code snippet, we set default value as 0.5. The lower this score values, the more appropriate the code snippets are for recommendation.

## 4 Case Study

We conducted an experimental case study on LFM, MAPO and LFM-OUPD. The case study is performed to investigate whether LFM-OUPD can assist programmers to figure out API usage patterns and locate code examples effectively.

### 4.1 Setup

To establish the current graph dataset, we focus our effort on one programming language, Java, and select one specific API library, Google Guava (see Footnote 1). This made our data collection and processing more relevant and manageable. We utilized dataset offered by Sawant and Bacchelli [16], which extracts method invocation and annotation references information about Google Guava API from 3013 projects.

LFM and LFM-OUPD are realized using Python language. LFM-OUPD implementation supports the recommendation of code snippets for a framework API implemented in Java. All experiments are carried out on a machine with CPU 2.7 GHz Intel Core i5, 8 GB RAM.

### 4.2 CacheBuilder Method Case

The CacheBuilder class[4] in Google Guava provides support to actually construct cache instances and set the desired features. It uses fluent style of building and provides various options of setting properties on the cache.

We conducted a comparison experiment study on LFM, MAPO and LFM-OUPD. The study aims to investigate whether LFM-OUPD can assist programmers locate API pattern and code snippet examples of interest efficiently. LFM

[4] https://google.github.io/guava/releases/17.0/api/docs/com/google/common/cache/CacheBuilder.html.

**Table 1.** API usage patterns returned by LFM

| PID | Cohesiveness | Availability | Representative | Mapped API usage |
|---|---|---|---|---|
| 1 | 0.82 | 0 | 1 | Non-representative API usage |
| 2 | 0.84 | 0 | 0.83 | Non-representative API usage |
| 3 | 0.62 | 0.15 | 0.67 | CacheBuilder set testing timed eviction |
| 4 | 0.44 | 0.1 | 0.87 | CacheBuilder set size-based eviction |
| 5 | 0.25 | 0 | 0.92 | Non-representative API usage |

returns 2484 API usage patterns from the API method-based sequences graph. There are 16 API usage patterns containing target API method. Table 1 illustrates the performance of top 5 API usage patterns. Obviously, their performance is not satisfying enough to support recommendation. Moreover, only two of them are valid usage patterns that correctly demonstrate CacheBuilder usages. However, these two method call sequences still miss approaches compared with representative code snippets.

LFM-OUPD returns 1712 API usage patterns from the API method-based sequences graph. There are 76 API usage patterns containing target API method CacheBuilder.newBuilder(). Table 2 illustrates the performance of top 10 API usage patterns sorted by ranking scores.

**Table 2.** API usage patterns returned by LFM-OUPD

| PID | Cohesiveness | Availability | Representative | Mapped API usage |
|---|---|---|---|---|
| 1 | 0.82 | 0.1 | 1 | CacheBuilder set size-based eviction |
| 2 | 0.78 | 0.102 | 1 | CacheBuilder set timed eviction |
| 3 | 0.77 | 0.1 | 0.875 | CacheBuilder set testing timed eviction |
| 4 | 0.62 | 0.104 | 1 | CacheBuilder set reference-based eviction |
| 5 | 0.67 | 0.65 | 1 | Initialization mechanism from a CacheLoader |
| 6 | 0.66 | 0.34 | 1 | Initialization mechanism from a Callable |
| 7 | 0.54 | 0.08 | 1 | Mechanism of explicit removals |
| 8 | 0.52 | 0.07 | 1 | Mechanism of removal listeners |
| 9 | 0.08 | 0.01 | 0.67 | Sequence pattern contains developer's test code |
| 10 | 0.05 | 0.01 | 0.43 | Sequence pattern contains isolated approach |

Column 'Mapped API Usage' interprets the mapping between detected API usage patterns and their mapped usage implementation mechanisms. There are eight valid usage pattern candidates that correctly demonstrate CacheBuilder usages. Method call sequence candidates from pattern 1 to 4 demonstrate mechanism of setting eviction cache features with different requestor objects and attributes. Method call sequence of pattern 5 demonstrates mechanism of initialization from a CacheLoader. API usage pattern 6 demonstrates mechanism

of initialization from a Callable. API usage candidates 7 and 8 demonstrate mechanism of reference-based eviction.

Taking the mapped API usage 5 as an example, the corresponding method call sequence for 'initialization from a CacheLoader' usage is as follows.

```
com.google.common.cache.CacheBuilder.newBuilder
com.google.common.cache.CacheBuilder.maximumSize
com.google.common.cache.CacheBuilder.build
com.google.common.cache.LoadingCache.get
```

Code snippet of 'CacheBuilder.newBuilder' for this pattern recommended by LFM-OUPD is illustrated in Fig. 6. To better understand this API usage, method calls appearing in pattern 5 are marked in red lines. This is a code snippet helping programmers to study how to initialize a cache using a CacheLoader class.

```
01 LoadingCache<Key, Graph> graphs = CacheBuilder.newBuilder()
02         .maximumSize(1000)
03         .build(
04             new CacheLoader<Key, Graph>() {
05                 public Graph load(Key key) throws AnyException {
06                     return createExpensiveGraph(key);
07                 }
08             });
09
10
11 try {
12     return graphs.get(key);
13 } catch (ExecutionException e) {
14     throw new OtherException(e.getCause());
15 }
```

**Fig. 6.** Code snippet of 'CacheBuilder.newBuilder'

Table 3 presents a comparison of other two approaches and LFM-OUPD. As the table shows, for the given 8 API usages, LFM-OUPD could detect and recommend all related API usage patterns based on user query, while the other two approaches' recommended results are subset of usage patterns returned by LFM-OUPD. Thus, our approach performs better than LFM and MAPO.

The case study results demonstrate that LFM-OUPD is effective for assisting programmers to identify different API usages. Also, LFM-OUPD greatly help programmers pick up appropriate representative code examples transformed from API usage candidates with less selecting and modification effort.

Furthermore, comparing with LFM, LFM-OUPD approach enhances the cohesiveness, availability, representative metric dramatically and could offer more representative API usage patterns. This can help programmers figure out more usage examples of target method.

**Table 3.** API usage patterns returned by three approaches

| PID | Mapped API usage | LFM-OUPD | LFM | MAPO |
|---|---|---|---|---|
| 1 | CacheBuilder set size-based eviction | Y | Y | Y |
| 2 | CacheBuilder set timed eviction | Y | N | Y |
| 3 | CacheBuilder set testing timed eviction | Y | Y | Y |
| 4 | CacheBuilder set reference-based eviction | Y | N | N |
| 5 | Initialization mechanism from a CacheLoader | Y | N | Y |
| 6 | Initialization mechanism from a Callable | Y | N | Y |
| 7 | Mechanism of explicit removals | Y | N | Y |
| 8 | Mechanism of removal listeners | Y | N | N |

## 5 Conclusion

In this work, an approach, LFM-OUPD, is proposed to recommend proper code examples for assisting programmers learning API more efficiently. LFM-OUPD is a graph-based approach, using community detecting approach to detect API usage patterns, which exits as high weight and density sub-graph in the API method call sequence graph. The detailed structural links among API method elements in conceptual source codes are captured and stored as graph structure. In LFM-OUPD, a mining algorithm based on LFM is presented to detect candidates of different API usage patterns. Also three appropriateness metrics are provided to assist ranking and recommending usage pattern. The effectiveness of LFM-OUPD is evaluated through a comparison case study. The study results demonstrate that, given an API method, LFM-OUPD can detect its various usages and recommend reliable code examples.

## References

1. Aeschlimann, M., Baumer, D., Lanneluc, J.: Java tool smithing extending the eclipse java development tools. In: Proceedings of the 2nd EclipseCon (2005)
2. Baumes, J., Goldberg, M.K., Krishnamoorthy, M.S., Magdon-Ismail, M., Preston, N.: Finding communities by clustering a graph into overlapping subgraphs. IADIS AC **5**, 97–104 (2005)
3. Bosu, A., Carver, J.C., Bird, C., Orbeck, J., Chockley, C.: Process aspects and social dynamics of contemporary code review: insights from open source development and industrial practice at microsoft. IEEE Trans. Softw. Eng. **42**, 302–321 (2016)
4. Clauset, A.: Finding local community structure in networks. Phys. Rev. E **72**(2), 026132 (2005)
5. Gregory, S.: An algorithm to find overlapping community structure in networks. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) PKDD 2007. LNCS, vol. 4702, pp. 91–102. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74976-9_12

6. Ko, A.J., Myers, B., Aung, H.H.: Six learning barriers in end-user programming systems. In: IEEE Symposium on Visual Languages and Human Centric Computing, pp. 199–206 (2004)
7. Lancichinetti, A., Fortunato, S., Kertész, J.: Detecting the overlapping and hierarchical community structure in complex networks. New J. Phys. **11**(3), 033015 (2009)
8. Malliaros, F.D., Vazirgiannis, M.: Clustering and community detection in directed networks: a survey. Phys. Rep. **533**(4), 95–142 (2013)
9. Mar, L.W., Wu, Y.C., Jiau, H.C.: Recommending proper API code examples for documentation purpose. In: 2011 18th Asia Pacific Software Engineering Conference (APSEC), pp. 331–338 (2011)
10. Newman, M.E., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E **69**(2), 026113 (2004)
11. Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M.: Prompter: a self-confident recommender system. In: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 577–580. IEEE (2014)
12. Radevski, S., Hata, H., Matsumoto, K.: Towards building API usage example metrics. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 619–623. IEEE (2016)
13. Robillard, M.P.: What makes APIs hard to learn? Answers from developers. Softw. IEEE **26**(6), 27–34 (2009)
14. Saied, M.A., Abdeen, H., Benomar, O., Sahraoui, H.: Could we infer unordered API usage patterns only using the library source code? In: 2015 IEEE 23rd International Conference on Program Comprehension (ICPC), pp. 71–81 (2015)
15. 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. IEEE (2015)
16. Sawant, A.A., Bacchelli, A.: A dataset for API usage. In: Proceedings of the 12th Working Conference on Mining Software Repositories, pp. 506–509. IEEE Press (2015)
17. Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D.: Mining succinct and high-coverage API usage patterns from source code. In: 2013 10th IEEE Working Conference on Mining Software Repositories (MSR), pp. 319–328 (2013)
18. Wang, L., Fang, L., Wang, L., Li, G., Xie, B., Yang, F.: APIExample: an effective web search based usage example recommendation system for Java APIs. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 592–595 (2011)
19. Wu, Y.C., Mar, L.W., Jiau, H.C.: CoDocent: support API usage with code example and API documentation. In: 2010 Fifth International Conference on Software Engineering Advances (ICSEA), pp. 135–140 (2010)
20. Xie, J., Kelley, S., Szymanski, B.K.: Overlapping community detection in networks: the state-of-the-art and comparative study. ACM Comput. Surv. (CSUR) **45**(4), 43 (2013)
21. Zagalsky, A., Barzilay, O., Yehudai, A.: Example overflow: using social media for code recommendation. In: 2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE), pp. 38–42 (2012)
22. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: mining and recommending API usage patterns. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 318–343. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03013-0_15