The Journal of Systems and Software 129 (2017) 127–139

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

API usage pattern recommendation for software development

Haoran Niu*, Iman Keivanloo, Ying Zou

Department of Electrical and Computer Engineering, Queen's University, Kingston, Ontario, Canada

ARTICLE INFO

Article history:
Received 1 February 2015
Revised 7 July 2016
Accepted 14 July 2016
Available online 16 July 2016

Keywords:
Usage pattern
Object usage
Clustering

ABSTRACT

Application Programming Interfaces (APIs) facilitate pragmatic reuse and improve the productivity of software development. An API usage pattern documents a set of method calls from multiple API classes to achieve a reusable functionality. Existing approaches often use frequent-sequence mining to extract API usage patterns. However, as reported by earlier studies, frequent-sequence mining may not produce a complete set of usage patterns. In this paper, we explore the possibility of mining API usage patterns without relying on frequent-pattern mining. Our approach represents the source code as a network of object usages where an object usage is a set of method calls invoked on a single API class. We automatically extract usage patterns by clustering the data based on the co-existence relations between object usages. We conduct an empirical study using a corpus of 11,510 Android applications. The results demonstrate that our approach can effectively mine API usage patterns with high completeness and low redundancy. We observe 18% and 38% improvement on F-measure and response time respectively comparing to usage pattern extraction using frequent-sequence mining.

1. Introduction

In recent years, the market of mobile applications has experienced a tremendous success (VisionMobile, 2013). The existence of Application Programming Interfaces (APIs) is one of the important factors contributing to the success of app development economy (Linares-Vásquez et al., 2013). Quite often, developers encounter unfamiliar or complex APIs that need to be combined and programmed in a certain way (Scaffidi, 2005). Earlier studies (Acharya et al., 2007; Bruch et al., 2008) show that API usage patterns are useful resources for developers in programming with unfamiliar APIs.

An API usage pattern is a sequence of method calls required to implement a functionality where participant methods belong to multiple API classes (Zhong et al., 2009). For example, if we want to read the input data from a remote device using `BluetoothSocket` class, the usage pattern P_b (Fig. 1) gives us a complete picture of the solution. At first, we have to create a `Blu e toothSocket` to connect to a device by calling the `createRfcommSocketToServiceRecord()` method of `BluetoothDevice` API. Next, we have to call `BluetoothSocket.connect()` method to establish a connection to the remote device. Once the socket is established, we can open the IO stream by calling `BluetoothSocket.getInputStream()` method, and then read the input stream data using `InputStream.read()`. Finally, we close the socket by calling the `close()` method of `BluetoothSocket` API. In such scenarios, a usage pattern, such as P_b in Fig. 1, helps developers in programming with unfamiliar APIs. A usage pattern provides a blueprint that shows the required steps and the relevant APIs and method calls.

Due to the lack of documentation, API usage pattern recommendation systems have been devised to support developers to learn how to use APIs (Acharya et al., 2007; Bajracharya et al., 2006; Buse and Weimer, 2012). A recommendation system mines a list of usage patterns that potentially answer developer's queries. Given a query which consists of an API name (*e.g.,* API class or method name), the mining process exploits a corpus of source code to automatically extract the potential usage patterns for the query. Moreover, a full-fledged recommendation system ranks the extracted usage patterns based on certain criteria, such as the popularity of the patterns within the corpus (Wang et al., 2013) or the relevance between the context of the patterns and a query (Zhong et al., 2009).

Since APIs might be designed to accomplish different functionalities (Mendez et al., 2013), an ideal mining process should identify at least one usage pattern for each functionality of an API (Grahne and Zhu, 2003). As noted by Grahne and Zhu (2003), there are two major threats to the success of an automatic API usage pattern extraction method: (1) Incompleteness and (2) redundancy in the recommended result set. Incompleteness means that a mining approach might not be able to mine all possible usage patterns for

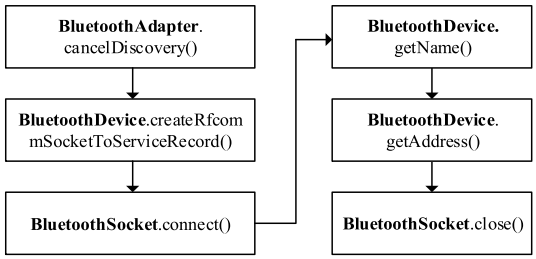
* Corresponding author.
E-mail addresses: 13hn@queensu.ca, hit.haoran@gmail.com (H. Niu), iman.keivanloo@queensu.ca (I. Keivanloo), ying.zou@queensu.ca (Y. Zou).

http://dx.doi.org/10.1016/j.jss.2016.07.026
0164-1212/

Method GetInformation()

```
public void GetInformation(BluetoothDevice mmDevice) {
    ...
mAdapter.cancelDiscovery();
BluetoothSocket mmSocket = mmDevice.createRfcommSocketToServiceRecord(...);
// Make a connection to the BluetoothSocket
try {
mmSocket.connect();
} catch (Exception e) {
    Log.e(TAG, "Connection to " + mmDevice.getName() + " at "
     + mmDevice.getAddress() + " failed:" + e.getMessage());
// Close the socket
mmSocket.close(); }
}
```

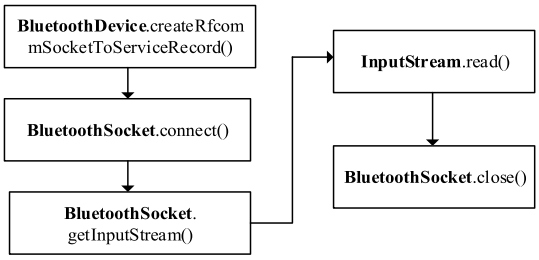
Usage Pattern P_a



Method ReadInput()

```
protected void ReadInput(BluetoothDevice device){
try {
//Create a Socket connection
BluetoothSocket socket = device.createRfcommSocketToServiceRecord( … );
socket.connect();
InputStream tmpIn = socket.getInputStream();
byte[] buffer = new byte[1024];
int bytes = tmpIn.read(buffer);
} catch (IOException e) {
Log.e("READINPUT", "Temp socket in created: " + e.getMessage());  }
socket.close();
}
```

Usage Pattern P_b



Method WriteOutput()

```
protected void WriteOutput(BluetoothDevice device) {
try {
//Create a Socket connection
BluetoothSocket socket = device.createRfcommSocketToServiceRecord( … );
socket.connect();
OutputStream outputStream = socket.getOutputStream();
outputStream.write(new byte[] { (byte) 0xa0, 0, 7, 16, 0, 4, 0 });
} catch (IOException e) {
Log.e("WRITEOUTPUT", "socket in created: " + e.getMessage());  }
finally {
if (socket != null) { socket.close();  return ; }}
}
```

Usage Pattern P_c

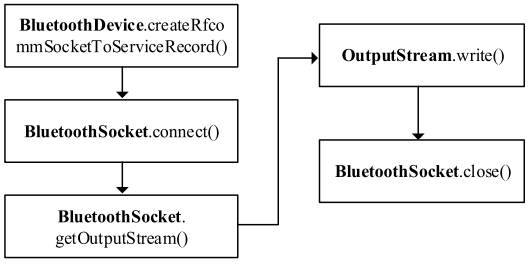


Fig. 1. Methods presenting the usages of `BluetoothSocket` class and extracted API usage patterns.

an API. Redundancy occurs when several similar API usage patterns for a single functionality are reported. Redundancy incurs more efforts for developers to find the API usage patterns of interest in the result set.

Earlier studies show that frequent-sequence mining (Wang and Han, 2004) are useful for mining API usage patterns. In such approaches, a support threshold, as part of the configuration of the frequent-sequence mining, is used to specify the minimum number of times that a set of methods are used together in order to be considered as a usage pattern. A low support threshold can achieve more completeness since more usage patterns are discovered; however, it may result in high redundancy (Wang and Han, 2004). A high support threshold can reduce redundancy but lead to low completeness since it may overlook some of the usage patterns. In practice, the threshold value helps balance the trade-off between incompleteness and redundancy.

An imbalanced usage occurs when one of the usage patterns of an API is considerably more frequent in a corpus than the other usage patterns. This characteristic reduces the chance of the other API usage patterns to survive from the support threshold of a frequent-sequence miner. Hence, it negatively affects the completeness of the result set (Wang et al., 2013). An earlier study by Mendez et al., (2013) reports that imbalanced usage is common in Java ecosystem. Similarly, our preliminary study on 11,510 Android applications shows the majority of API classes used in Android mobile application development experience imbalanced usage. Hence, existing approaches using frequent-sequence mining might not be suitable for mining API usage patterns from Android applications.

In this paper, we aim to automatically extract usage patterns without using frequent-sequence mining. The usage pattern mining process in our approach avoids being affected by the imbalanced usage behavior observed in Java ecosystem (Mendez et al., 2013). Hence, we expect to improve the completeness of the result set.

Given an API method as a query, our approach automatically recommends a set of relevant API usage patterns. Our approach is based on the concept of object usage (Monperrus and Mezini, 2012), which captures method calls invoked only on an object of a single API class. For example, the method calls belonging to `mmDevice` object of class `BluetoothDevice` in method `GetInformation()` shown in Fig. 1 can be summarized as an object usage, denoted as O_2 listed in Table 1. In this approach, an API usage pattern can be modeled as a set of object usages. For example, the underlying API usage pattern of method `GetInformation()`is P_a as shown in Fig. 1, and can be represented as a set of object usages $\{O_1, O_2, O_3\}$ as listed in Table 1.

Table 1
Object usages extracted from method `GetInformation()` in Fig. 1.

Objectusage	API class	Participating methods
O_1	Bluetooth adapter	• cancelDiscovery()
O_2	Bluetooth device	• createRfcommSocket ToServiceRecord() • getAddress() • getName()
O_3	Bluetooth socket	• close() • connect()

We extract the object usages that match with the API described in a query, and capture their co-existence relations with other object usages within the same methods in the corpus. Then, we model the extracted object usages and their co-existence relations with others as a network. Finally, we identify API usage patterns by clustering similar object usages in the network.

This paper makes the following contributions:

- It proposes an approach to improve the mining of less frequently used API usage patterns. Our approach studies the co-existence relations among object usages and represents the relationship using a network. To reduce the redundancy of resulting usage patterns, we identify a representative object usage from each cluster where object usages are internally similar.
- It evaluates our approach using a corpus of 11,510 Android applications crawled from Google Play. The evaluation results show that our approach can effectively mine API usage patterns with high completeness and low redundancy. We observe an improvement of 18% and 38% on F-measure and response time respectively comparing to a frequent-sequence miner.

Organization of the rest of the paper. Section 2 describes the related concepts and challenges in mining API usage patterns. Section 3 provides the details of our approach. Section 4 explains case study design. Section 5 presents our case study results. Section 6 discusses threats to validity. Finally, related work and conclusion are presented in Sections 7 and 8.

2. Definitions

In this section, we describe the basic concepts related to our research that are usage pattern and object usage. Furthermore, we elaborate on the threats to the success of mining and recommending API usage patterns using an example.

2.1. Usage pattern

It is common for an API class to be used in different contexts to realize various functionalities (Wang et al., 2013). For example, class `BluetoothSocket` from `Bluetooth` package of Android development kit can be used to implement three functionalities related to Bluetooth technology. One functionality is shown in method `GetInformation()` in Fig. 1 where the information of a remote device can be retrieved after establishing a Bluetooth socket connection. Methods `ReadInput()` and `WriteOutput()` show the other two functionalities implemented using `BluetoothSocket`, which read or write data via Bluetooth channels. As highlighted in the methods listed in Fig. 1, each of the three pieces of functionality involves multiple method calls from various classes.

A usage pattern summarizes a sequence of method calls required to implement a functionality where the method calls belong to multiple API classes (Zhong et al., 2009). Formally, a usage pattern can be defined as $P = \{ m_{a,1} \sim m_{b,2} \sim \cdots \sim m_{n,i} \}$ $(i \geq 2)$, where mn, i denotes the i_{th} method call in an implementation of the certain functionality; the number of the method calls is at least 2 since the implementation of a functionality needs the cooperation of different method calls; n refers to an API class that method $m_{n,i}$ belongs to and the number of participant API classes depends on the functionality. In addition, the sequential rule "$\sim$" is used to represent the order of usage. For example, the rule A$\sim$B between method calls A and B means that method A is called before method B in the usage scenario.

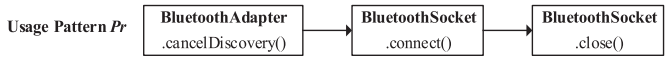


Fig. 2. A possible output of an API usage pattern mining system for *query* = {*BluetoothSocket*, *connect*}.

For example, the functionality for reading data using `BluetoothSocket` shown in method `ReadInput()` (Fig. 1) involves five method calls: `createRfcommSocke-tToServiceRecord()`, `connect()`, `getInputStream()`, `read()` and `close()`, the five method calls are from three different API classes: `BluetoothDevice`, `BluetoothSocket` and `InputStream`. Similarly, the method calls involved in the functionalities of getting information and writing output are highlighted in the corresponding methods shown in Fig. 1. Therefore, the three functionalities related to class `BluethoothSocket` can be summarized as three usage patterns P_a, P_b, and P_c shown in Fig. 1. In this graphical presentation, each rectangle represents a method call involved in a usage pattern. The method name of method call is placed at the bottom following the class name at the bottom of the rectangle.

2.2. Object usage

Object usage is defined as a set of method calls invoked on a single object of a given API class in a method within a corpus (Monperrus and Mezini, 2012). Formally, an object usage of an API class C is defined as a set of method calls, *i.e.*, $O = \{ m_1, m_2, \cdots, m_n \}$ where m_n represents a method call; n is the total number of method calls invoked on an object of API class C in a method. Object usage does not consider the sequence of the involved method calls in an object usage. For example, object usage O_2 in Table 1 is extracted from method `GetInformation()` shown in Fig. 1. Object usage O_2 corresponds to object `mmDevice` of class `BluetoothDevice`. Table 1 shows three object usages extracted from method `GetInformation()` in Fig. 1.

Being different from a usage pattern that includes method calls from multiple API classes, an object usage contains method calls from a single API class. Therefore, we can model a usage pattern as a set of object usages. For example, usage pattern P_a in Fig. 1 can be represented as a set of object usages $\{O_1, O_2, O_3\}$ as listed in Table 1.

2.3. Challenges in mining API usage patterns

In the context of mining and recommending usage patterns, a query specifies the search criteria. It contains an API class and a method of the API class. The output of the query is a ranked list of API usage patterns relevant to the query (Zhong et al., 2009). For example, when a developer searches the method `connect()` in `BluetoothSocket` class, *i.e.*,*query* = {*BluetoothSocket*, *connect*}, the potential answers are P_a, P_b, and P_c, as shown in Fig. 1.

A frequent-sequence mining method (*e.g.*, (Wang et al., 2013)) extracts three usage patterns P_b (Fig. 1), P_c (Fig. 1) and P_r (Fig. 2) as the answer to the query {*BluetoothSocket*, *connect*}. In this case, we observe redundancy among the recommended usage patterns.

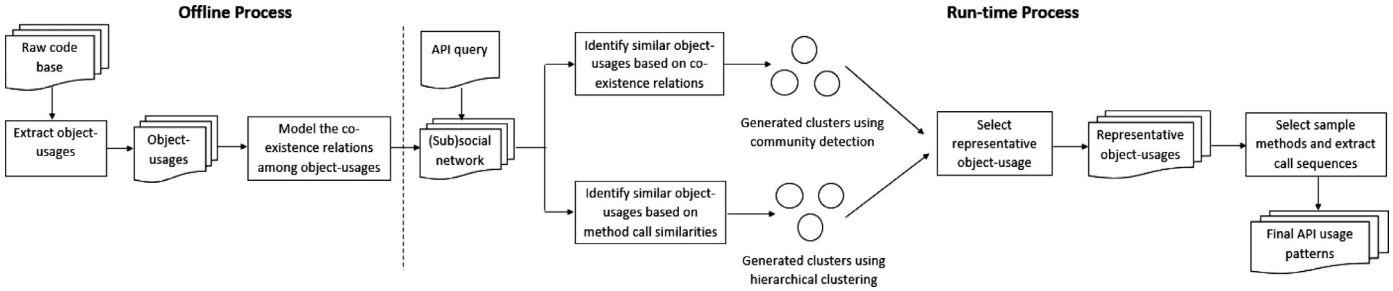


Fig. 3. Overview of our API usage pattern recommendation approach based on object usage clustering.

P_r is redundant since the underlying pattern is already covered by P_b and P_c in the result set (Wang et al., 2013). Furthermore, usage pattern P_a (Fig. 1) which represents the functionality of fetching information is not identified by the mining approach. This functionality is less frequent than the two other functionalities covered by P_a and P_b. Failure to identify a known usage pattern reduces the completeness of the result set since each functionality related to `BluetoothSocket` class should be covered by at least one of the mined usage patterns (Wang et al., 2013). We conjecture that an approach that does not use frequent-sequence mining might be able to successfully extracts P_a even if P_a is less common than P_b and P_c due to the imbalanced usage behavior (Mendez et al., 2013).

3. The proposed approach

Fig. 3 illustrates an overview of our approach. The approach contains two processing phases. The first phase focuses on the processing of the corpus, and is performed offline. The second phase occurs at run-time when a query is issued by a developer. The run-time process contains two major steps: clustering similar object usages and recommending API usage patterns. Specifically, we extract object usages from the methods in the corpus. If the object usages come from the same method, we consider such object usages have co-existence relations. To identify API usage patterns, we use two clustering algorithms to group the object usages based on co-existence relations or method call similarity. To avoid having redundant answers in the final result set, we select a representative answer from each cluster of object usages. The representative object usages are mapped to API usage patterns which would be recommended to developers. We discuss the details of each phase in the following subsections.

3.1. Modeling co-existence relations

We analyze the corpus to capture the co-existence relations among object usages within the same method. We further model the object usages and their co-existence relations as a network, *i.e.*, $G = (V, E)$, where V is a set of object usages, and E is a set of links that represent the co-existence relations between object usages. Each link is labeled with a weight. The weight of a link represents the number of times that co-existence relations occur between object usages. The network can be built using static analysis on the methods in a corpus. Similar to the approach proposed by Monperrus and Mezini (2012), we analyze the methods to extract method calls on objects to identify object usages. Fig. 4 shows an example of such network if we assume the corpus is built from the three methods related to `BluetoothSocket` class as discussed in Section 2. Object usages O_1,O_2 and O_3 are extracted from method `GetInformation()` and are listed in Table 1. Object usages O_4, O_5 and O_6 are extracted from method `ReadInput()`. The object usages extracted from method `WriteOutput()` are denoted by O_4, O_7, and O_8. The weight of links is labelled in the sample network shown in Fig. 4, each weight is 1 since the related object usages co-occur only once in our sample data.

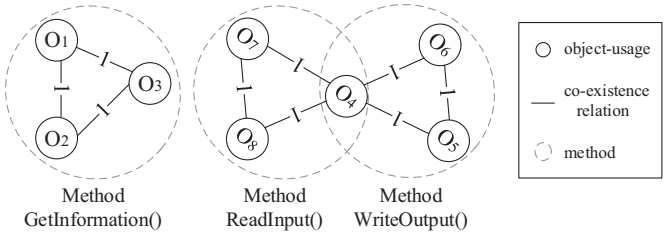


Fig. 4. An example for the network created using the co-existence relations among object usages in methods `GetInformation()`, `ReadInput()`, and `WriteOutput()` shown in Fig. 1.

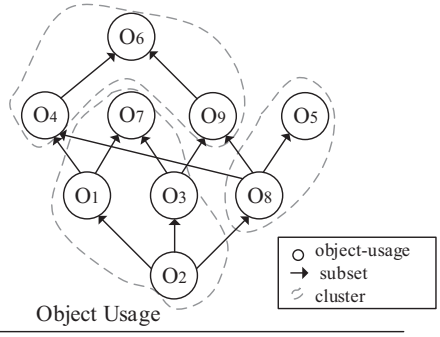
3.2. Clustering similar object usages

In our approach, object usages are eventually mapped to API usage patterns. The redundancy in the object usages can lead to the redundancy in the final results of API usage patterns. In this section, we discuss the process of clustering similar object usages to solve the redundancy issue.

Given a query, we first extract the relevant object usages that contain the API method specified in the query. For example, when a query is the method `connect()` in class `BluetoothSocket` as discussed in Section 2, *i.e.*, *query* = {*BluetoothSocket*, *connect*}, our approach finds all object usages that contain the method `connect()` in class `BluetoothSocket`. Therefore, we can extract a (sub)social network that contains a set of candidate object usages and their co-existence relations with others.

However, the candidate object usages suffer from the redundancy issue (Mendez et al., 2013), which eventually leads to redundancy in the final result set of API usage patterns. For example, as shown in Fig. 5, there are 9 candidate object usages relevant to the query, {*BluetoothSocket*, *connect*}. If we include one usage pattern for each candidate object usage, we will achieve about 67% duplication in the result according to the duplication measure used in Wang et al. study (Wang et al., 2013) since there are just 3 common usage patterns for *BluetoothSocket* class shown in Fig. 1. To address this concern and decrease the duplication ratio in the result set, we apply clustering on the candidate object usages to identify similar object usages. There exist two forms of similarity between object usages: co-existence relations and method call similarity (Mendez et al., 2013). Therefore, our approach uses two clustering techniques to summarize the similar object usages from the two similarity points of view.

In the following subsections, we describe the details of our two clustering approaches.



No.	Object Usage
O1	connect(),getInputStream()
O2	connect()
O3	connect(),getOutputStream()
O4	close(),connect(),getInputStream()
O5	close(),connect(),getRemoteDevice()
O6	close(),connect(),getInputStream(),getOutputStream()
O7	connect(),getInputStream(),getOutputStream()
O8	close(),connect()
O9	close(),connect(),getOutputStream()

Fig. 5. An example object usage map for `BluetoothSocket` class.

3.2.1. Clustering similar object usages based on co-existence relations

As pointed out by Newman and Girvan (2004), a network has *community structure* if the nodes can be grouped into subgroups where the nodes are densely connected internally. In our approach, we identify similar object usages by partitioning object usages into communities. We use a *modularity* measure, as denoted by M, to detect communities within a network of object usages. Modularity M measures whether the number of intra- community links is more than that of inter-community. Therefore, a high modularity value indicates a good partition of object usages into communities in a network (Clauset et al., 2004).

$$\mathrm{M} = \sum (e_{ii} - a_i^2) \tag{1}$$

Let e_{ij} *be a fraction of inter-community links that connect object usages in community* C_i *to object usages in community* C_j*; let* $a_i = \sum_j e_{ij,}$ *and* a_i *represents the fraction of intra-community links that connect object usages within community* C_i*. In our study, the weight of the links between object usages is used to compute* e_{ij}.

It is not feasible in practice to search exhaustively all possible partitions of nodes to find the best one for a large network (Newman, 2004). Similar to a greedy optimization algorithm proposed by Clauset et al., (2004) to tackle such complex problem, we use a greedy optimization algorithm listed in Algorithm 1 to detect communities among object usages in the network. In an initial state, each object usage is a sole member of one community. We attempt to join communities in pairs by observing the impact of the joining activity on the modularity value, *i.e.* changes of M, ΔM. We aim to join two communities when joining the two communities can achieve the greatest increase in the modularity value. In the initial state, we compute the initial values of modularity using Eq. (1), and the initial values in ΔM matrix are computed using Eq. (2). We select the largest modularity change value, ΔM_{ij}, from ΔM matrix, and merge the corresponding communities C_i and C_j. Then, we update matrix ΔM. We repeat the process until the maximum modularity is achieved. Using the greedy algorithm, our approach produces the optimal community partition (Clauset et al., 2004).

$$\Delta M_{ij} = \begin{cases} 1/2m - k_i k_j/(2m)^2 & \text{if } C_i,\ C_j \text{ connected} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

where m is the weight of links between object usages in a network; and k_i *and* k_j *are the number of incident links on* C_i *and* C_j *respectively.*

Following our running example on the `BluetoothSocket` API, we apply Algorithm 1 to detect communities on the object usage network which contains the object usages relevant to the query {*BluetoothSocket*, *connect*} listed at the bottom of Fig. 5 and also includes other object usages having co-existence relations with the nine object usages. We obtain three clusters (communities) for object usages related to the query as: object usages O_1, O_4, O_6 and O_7, object usages O_2, O_5 and O_8, and object usages O_3 and O_9.

3.2.2. Clustering similar object usages based on method-call similarity

Object usages of a specific API class can be subsets of each other or share a large number of method calls with each other (Mendez et al., 2013). Object usage map is proposed by Mendez et al., (2013) to represent the subset relations among object usages. In an object usage map, each node is an object usage and the edges correspond to a subset relation between two object usages. More specifically, if all the methods in an object usage O_x are contained in an object usage O_y, there is an arrow from O_x to O_y. et al. Fig. 5 shows an object usage map for the object usages relevant to the query {*BluetoothSocket*, *connect*}. As shown in Fig. 5, most of the relevant object usages are partially similar to each other. For example, O_3 is a subset of O_7 and O_9 since all methods included in O_3 are occurred in O_7 and O_9.

Due to the complexity of the graph, it is not trivial to group similar object usages (Mendez et al., 2013). Quite often, there could be hundreds of nodes in an object usage map related to a single API, *e.g.,* the `File` API in Java has 2166 usages (Mendez et al., 2013). To reduce the redundancy of object usages, we strive for grouping similar object usages together using hierarchical clustering on the object usage map.

To apply hierarchical clustering, each object usage is initially treated as a separate cluster. Then the pairs of clusters that have the minimum inter-cluster distances are selected to be merged. We calculate the inter-cluster distances using the complete linkage technique (Han and Kamber, 2006), *i.e.,* the distance between two clusters is taken from the maximum distance between any pairs of object usages in the two clusters. The distance among any pairs of object usages is computed using Euclidean metric. We represent each object usage as a vector, *i.e.*,$V_o = m_1,\ m_2,\ \ldots,\ m_i,\ \ldots,\ m_n$, where m_i represents a method; and n is the total number of methods in an API class. m_i is set to 1 when the object usage contains the method m_i; otherwise, m_i is set to 0. Then, the Euclidean distance between two object usages O_a and O_b can be calculated as $D(O_a,\ O_b) = \sqrt{\sum_{i=1}^{n} (m_i^a - m_i^b)^2}$, where m_i^x is the i_{th} item in object usage vector x.

In the hierarchical clustering process, it is important to decide when the merging process should end, *i.e.,* deciding the number of final clusters. The selection of the number of clusters should aim to maximize the dissimilarity between object usages in different clusters and minimize the number of resulting clusters. We apply the Gamma quality index (Guerra et al., 2012) to determine the proper number of clusters for a given set of object usages. The Gamma index compares the distance of object usage pairs, and a comparison is considered as consistent if a distance between a pair of object usages which do not belong to the same cluster is smaller than the distance between a pair of object usages belonging to the same cluster; otherwise, the comparison is inconsistent. Gamma quality index is defined in Eq. (3). The values of Gamma quality index are between −1 and 1. The maximum value of Gamma index

represents an optimal clustering.

$$G = \frac{S^+ - S^-}{S^+ + S^-} \quad (3)$$

where S^+ is the number of consistent comparisons while S^- is the number of inconsistent comparisons.

For the example shown in Fig. 5, Gamma quality index suggests three as the number of expected clusters for the hierarchical clustering process. As illustrated in Fig. 5, The three clusters identified using the hierarchical clustering are marked in the dashed lines, *i.e.,* object usages O_1, O_2, O_3 and O_7, object usages O_4, O_6 and O_9, and object usages O_5 and O_8.

The two clustering algorithms help us to summarize similar object usages into clusters. To reduce the redundancy of resulting usage patterns, we select one representative from each cluster identified by the two clustering algorithms to create the representative object usage set. As defined by Mendez et al., (2013), the *abundance* metric describes the number of times that an object usage is used in a corpus. The abundance value of an object usage indicates its popularity. The higher the abundance value is, the more frequently one object usage is used in the corpus. We select the object usage with the highest abundance value as the representative of a cluster. Based on the popularities of the object usages shown in Fig. 5, object usages O_4, O_5, and O_9 are selected as representatives for the clusters identified based on co-existence relations. Object usages O_1, O_4, and O_5 are selected as representatives for the clusters identified based on method-call similarity. Therefore, object usages O_1, O_4, O_5 and O_9 constitutes the representative object usage set.

In summary, hierarchical clustering algorithm groups object usages based on the textual method-call similarities. Hierarchical clustering performs well in reducing the duplication of object usages. However, it may group the object usages dealing with different functionalities together (*e.g.*, O_4 and O_9 in Fig. 5) and may affect the completeness of the final API usage patterns. Community-based clustering algorithms groups object usages into different communities that indicate different functionalities. It helps improve the completeness of the final results. Therefore, combining the two clustering algorithms in our approach can balance the duplication and completeness of final results.

3.3. Recommending API usage patterns

The representative object usages are intermediate answers since they are not exactly API usage patterns (the expected form of output from an API usage pattern mining approach). Therefore, in the last step, we have to map object usages to API usage patterns. We use the mapping approach suggested by Mishne et al., (2012) to convert object usages to concrete usage patterns. In this approach, for each candidate object usage, we search in the corpus of actual code snippets, and find a real code snippet that uses the object usage. Furthermore, we extract the call sequence of the identified code snippet, and use it as the final usage pattern. Then, we add an API usage pattern to the final result set for each candidate object usage.

The higher popularity of one usage pattern has, the more frequently the usage pattern is used. In other words, the more likely the usage pattern is an expected answer of a developer (Wang et al., 2013). We should place the usage patterns with higher popularities at the top of the recommendation list, which can help developers find the expected answer quickly. Therefore, we rank the final API usage patterns based on the popularities of the usage patterns. In our study, the popularity of a usage pattern is represented by the abundance value of the representative object usage contained in the usage pattern.

Table 2
A Summary of our Android application corpus.

Corpus	Size
Android applications	11,510
Extracted call sequences	5625,750
Contained classes	420,298
Extracted object usages	3087,133

4. Case study design

The goal of the case study is to evaluate the performance of our approach in (1) mining usage patterns and (2) ranking usage patterns for Android application development. All the experiments are conducted on a 2.5 GHz CPU Windows 7 machine with 2GB memory. In this section, we discuss the details of creating our corpus, experiment setup and describe comparison baselines.

4.1. Data gathering

To evaluate our approach, we create a large-scale corpus of Android applications. Table 2 provides a summary of the corpus. The corpus contains 11,510 active Android mobile applications covering 33 different application domains, such as media and education. The applications are collected from Google Play by downloading the top applications of each category.

An Android application is represented as an Android Application Package (APK), which contains the compiled code. Since we download Android applications from Google Play, we do not have access to the source code of the applications at large (Mojica et al., 2014). APKTool (APKTool) is a reverse engineering tool to unpack the compiled code of Android applications into smali format (Hoffmann et al., 2013), a plain text presentation of the compiled code. Each invocation statement in the source code is represented as a single line starting with the keyword “invoke” in the smali code. The invocation line contains the method name, calling variable, parameters, and return type of a method call. We process the data in three steps. First, we unpack APKs using APKTool to represent the code in smali format. Then, we extract the call sequences from each method body in the unpacked smali code files by identifying the “invoke” keyword. Finally, we generate object usages from the extracted call sequences, based on the definition of object usage. We extract 3,087,133 unique object usages for 420,298 API classes, which constitutes our corpus.

Although the obfuscated code (Vásquez et al., 2014) might exist in our corpus, the object usage retrieval for a given API query is a keyword-based matching process, the obfuscated code would not appear in the retrieval result. Therefore, the obfuscated code would not affect our experimental results.

4.2. Performance evaluation setup

To evaluate the effectiveness of our approach in terms of mining usage patterns, we need to identify the outcome of an ideal usage pattern mining approach for a set of API queries. The set of ideal usage patterns constitutes the “gold set” used for evaluation. In the following, we describe the process of selecting queries, creating the “gold set”, and baseline approaches.

4.2.1. Selecting queries

We adopt the automatic framework proposed by Bruch et al., (2008) to randomly select a set of queries from our corpus. In this framework, first a method is randomly selected from the corpus and then the method call sequence in the method is extracted. The method call sequence acts as the expected usage pattern. Then, a query is automatically generated for the expected usage pattern by

randomly selecting a method call from the expected usage pattern and extracting the class and the invoked method in the method call. The automatic process of selecting queries reduces the subjective judgment in the evaluation (Bruch et al., 2008).

4.2.2. Creating a "gold set"

We use the same approach as Wang et al., (2013) to build the "gold set" for API queries. First, we randomly select a set of API queries from the corpus. For each API query, we automatically extract all the call sequences involving the API mentioned in the query from the corpus. Finally, we manually identify a set of representative usage patterns among the extracted call sequences by referring to the documentation (*e.g.*, JavaDocs) provided by the API owners. The identification process was done with the first author who has one-year Android application development experience and then verified by the second author and a non-author curator independently who both have more than five years Java programming experience. The two verifiers agreed with about 90% of the usage patterns identified by the first author. The majority rule is used to decide whether placing the remaining usage patterns in the "gold set" or not. Finally, the identified usage patterns for the API queries constitute the "gold set".

The overall evaluation process is performed as follows. Each query is executed by our approach. The returned usage patterns are compared with the usage patterns in the "gold set". If the proposed approach fails to report any of the known usage patterns in the "gold set" for a query, the completeness decreases. If the proposed approach reports more than one suggestion for any of the known usage patterns in the "gold set", the redundancy increases. Finally, performance measures are computed based on the comparison between the recommended usage patterns and the ones in the "gold set".

4.2.3. Baseline approaches

First, we compare our approach with the baseline approach that uses frequent-sequence mining to extract API usage patterns. The baseline approach is implemented as described in UP-Miner (Wang et al., 2013). We consider UP-Miner as the first baseline approach since UP-Miner outperforms its successors in terms of high completeness and low redundancy (Wang et al., 2013). Second, we compare our approach with an online usage pattern recommendation engine, Codota (Codota). Codota is an online and commercial usage pattern recommendation engine for Android application development. Codota is based on searching methods crawled from the Internet, *e.g.*, StackOverflow, while our approach exploits bytecode from Android applications.

5. Case study results

In this section, we first present our experimental results to answer three research questions. For each research question, we present the motivation behind the question, analysis approach and a discussion on our findings. Then, we explore the usefulness of our approach by designing a preliminary user study.

5.1. Experimental results

RQ 1: What is the performance of our approach for mining API usage patterns?

Motivation. A high-quality API usage pattern miner should achieve high completeness and low redundancy (Wang et al., 2013). In addition, API usage pattern mining systems can be used as online services, *e.g.*, Codota.com, so they have to maintain a reasonable response time regardless of the size of the underlying data to be processed. In this research question, we evaluate whether our approach performs better than the baseline approach that uses frequent-sequence mining.

Table 3
Mapping Cliff's delta with Cohen's standards.

Cliff's delta	% of Non-overlap	Cohen's d	Cohen's standards
0.147	14.7%	0.20	small
0.330	33.0%	0.50	medium
0.474	47.4%	0.80	large

Approach. We compare our approach with a representative frequent-sequence-mining based approach (Wang et al., 2013). We use the baseline and our approach to identify API usage patterns from our corpus summarized in Table 2 for queries available in the "gold set" described in Section 4.2.1). We compare the outcome using completeness, duplication and F-measure measures (Wang et al., 2013). Completeness measure is defined in Eq. (4). The values of the measure range from 0 to 1, a higher value is preferred. A high completeness ensures a high recall. Duplication measure is specified in Eq. (5). The values of duplication measure range from 0 and 1. A lower value is desired. A high completeness and a low duplication can ensure that the precision value of the final results is high. In addition, we use F-measure to evaluate the overall quality between completeness and duplication. The F-measure is described in Eq. (6), and its output ranges between 0 and 1. Higher values of F-measure are preferred.

$$Completeness = \frac{\#unique\ patterns}{\#\ patterns\ in\ G} \tag{4}$$

$$Duplication = \frac{\#duplicated\ patterns}{\#\ patterns\ in\ M} \tag{5}$$

$$F-measure = \frac{2 * (1\ -\ Duplication) * Completeness}{(1\ -\ Duplication)\ +\ Completeness)} \tag{6}$$

where *G* r*epresents a set of usage patterns in the "gold set"; M represents a set of mined usage patterns to be evaluated; #unique patterns is the number of usage patterns representing different API functionalities in the pattern set M; #duplicated patterns is the number of remaining usage patterns after excluding the unique usage patterns from the return usage pattern set.*

We use Wilcoxon rank sum test (Sheskin, 2007) to determine if the observed difference in the performance of the frequent-sequence-mining based approach and our approach is significant. Wilcoxon rank sum test is a non-parametric test that does not hold assumption on the distribution of data. We conduct Wilcoxon rank sum test with 5% confidence level (*i.e.*, p-value < 0.05). We calculate Cliff's delta as the effect size (Romano et al., 2006) to quantify the importance of the difference between the two approaches. Cliff's delta estimates non-parametric effect sizes. It makes no assumptions of a particular distribution (Romano et al., 2006), and is reported to be more robust and reliable than Cohen's (1988). Cliff's delta represents the degree of overlap between two sample distributions (Romano et al., 2006). It ranges from -1 to $+1$. Cliff's delta is -1 if all selected values in the first group are larger than the second group; and Cliff's delta is $+1$ if all selected values in the first group are smaller than the second group. Cliff's delta is zero when two sample distributions are identical (Cliff, 1993). Cohen's standards (*i.e.*, small, medium, and large) are commonly used to interpret effect size. Therefore, we map the Cliff's delta to Cohen's standards using the percentage of non-overlap (Romano et al., 2006). The mapping between the Cliff's delta and Cohen's standards is shown in Table 3. Cohen (1992) states that a medium effect size represents a difference likely to be visible to a careful

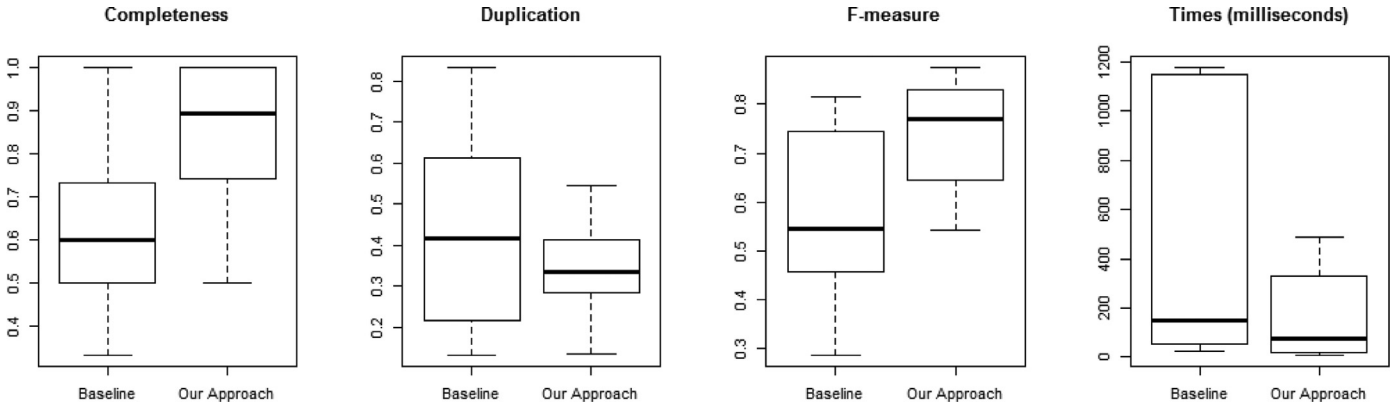


Fig. 6. Performance comparison between the baseline (based on frequent-sequence-mining) and our approach in terms of completeness, duplication, F-measure, and response time.

Table 4
The summary of the performance comparison between the baseline and our approach.

Metric	Approach	Mean	*p*-value	Cliff's delta	Effect size
Completeness	Baseline	0.62	**0.0003**	0.67	large
	Our approach	0.86			
Duplication	Baseline	0.43	0.3714	−0.17	small
	Our approach	0.34			
F-Measure	Baseline	0.56	**0.0009**	0.62	large
	Our approach	0.74			

observer, while a large effect is noticeably greater than medium. We also compare the response time of the two approaches in milliseconds.

Results. Fig. 6 summarizes the evaluation results of the frequent-sequence-mining approach and our approach in terms of the four studied performance measures. Table 4 lists the result of *p*-values and Cliff's delta for the three performance measures related to the result set quality (Wang et al., 2013). Our approach significantly outperforms the baseline in terms of completeness, with an average value of 0.86. The completeness values for 90% of the total number of queries have been improved by our approach. We also observe a non-significant improvement comparing to the frequent-sequence mining baseline from the duplication aspect. Finally, we observe that our approach can significantly improve the overall performance by achieving an average value of 0.74 in terms of F-measure with a large effect size in mining API usage patterns for Android application development as shown in Table 4. In addition, we found that our approach completes the mining process 38% faster than the baseline approach as shown in Fig. 6.

In our approach, clustering based on co-existence relations helps to improve the completeness of the final results, and clustering based on method-call similarities mainly aims to reduce the duplication of the final results. As shown in Table 4, the effect size for the completeness improvement is large while the effect size in terms of duplication is small. Therefore, the clustering algorithm based on co-existence relations (*i.e.,* community-based clustering) contributes more than clustering based on method-call similarities (*i.e.,* hierarchical clustering) in achieving better performance for mining API usage patterns. However, the two clustering algorithms are complementary to each other to achieve significant improvement in terms of F-measure.

We observe 18% improvement on F-measure and 38% faster response time comparing to a frequent-sequence-mining approach in mining API usage patterns for Android application development.

RQ 2: Is our approach effective in mining usage patterns for APIs experiencing imbalanced usage?

Motivation. Existing approaches for identifying API usage patterns mainly use the frequent-sequence mining technique (Wang et al., 2013). In such approaches, a support threshold value is needed to determine the minimum level of popularity of a pattern (Wang and Han, 2004). Therefore, the mining process may overlook part of the usage patterns of APIs with imbalanced usage (Mendez et al., 2013). Incomplete result set reduces the chance for developers to leverage existing code to implement the desired pieces of functionality (Wang et al., 2013). Our approach does not use the frequent-sequence mining. Therefore we conjecture that our approach outperforms the existing approaches in recommending API usage patterns for APIs that are experiencing imbalanced usage (Mendez et al., 2013).

Approach. In our research context, there exist two forms of imbalanced usage of APIs: (1) Some APIs are less frequent than other APIs; and (2) some usage patterns of a specific API are less frequent than the other usage patterns of the same API. We use *abundance* and *dominance* metrics to describe the characteristics of imbalanced usage in APIs (Mendez et al., 2013). Abundance describes the number of times that the object usages containing the API class and method specified in a query (relevant object usages) occur in the corpus. Dominance counts the amount of the relevant object usage instances that belong to its dominant object usage (*i.e.,* the most popular relevant object usage). We split the queries used in RQ1 into two groups: (1) A group with high abundance; and (2) a group with low abundance. If the abundance value of an API is more than the median, it belongs to the group of high abundance group; otherwise, it is classified into the group of lower abundance. Similarly, we apply the same criterion to obtain the two groups with high or low dominance level. We compare F-measures (Eq. (6)) of the frequent-sequence-mining based approach (Section 4.2.3) with our approach to test the following null hypothesis:

H_{01}: There is no difference between the performance of our approach and the frequent-sequence-mining based approach in mining usage patterns for APIs with low abundance.

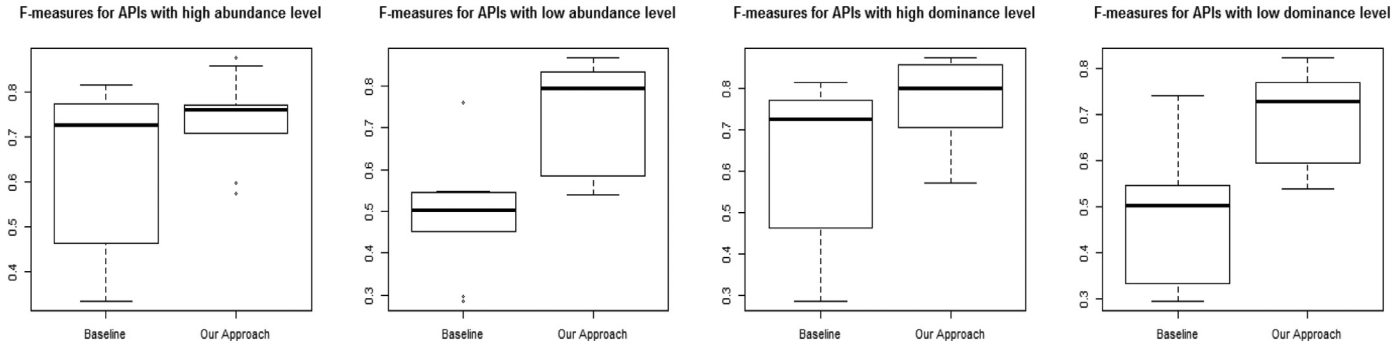


Fig. 7. Performance comparison between the baseline (based on frequent-sequence-mining) and our approach against APIs with different abundance or dominance level.

Table 5
A summary of performance comparison between UP-Miner and our approach against different API groups.

API groups	Approach	Mean	*p*-value	Cliff's delta	Effect size
High abundance	Baseline	0.63	0.272	0.30	small
	Our approach	0.74			
Low abundance	Baseline	0.49	**0.001**	0.86	large
	Our approach	0.73			
High dominance	Baseline	0.68	**0.023**	0.51	large
	Our approach	0.77			
Low dominance	Baseline	0.50	**0.009**	0.89	large
	Our approach	0.74			

H_{02}: H_{02}: There is no difference between the performance of our approach and the frequent-sequence-mining based approach in mining usage patterns for APIs with high dominance.

We apply Wilcoxon rank sum test (Sheskin, 2007) with 5% confidence level (*i.e.*, p-value < 0.05) to evaluate if the performance of the two approaches are significantly different. If there is a statistical significance, we can reject the null hypothesis, and conclude that the performances of our approach and UP-Miner are significantly different in recommending usage patterns for APIs with low abundance or high dominance. Then we calculate Cliff's delta as the effect size (Romano et al., 2006) to quantify the importance of the difference between the two approaches. Finally, we compare the F-measure values achieved by our approach and UP-Miner to decide the one that performs better in mining less frequently used usage patterns.

Results. Figure 7 summarizes the results of the F-measure study on mining usage patterns for the APIs with different abundance or dominance levels. In this research question, we are particularly interested in the low abundance and high dominance groups since they include the queries related to the APIs with the two forms of imbalanced usage (Mendez et al., 2013). As shown in Fig. 7, the F-measure of our approach is higher than that of frequent-sequence-mining baseline in answering queries related to APIs with imbalanced usage (*i.e.*, high or low abundance). As shown in Table 5, the F-measure improvement can be 24% for APIs with low abundance. In addition, the p-value of the performance comparison in mining usage patterns for API with low abundance is significant, so we can reject the null hypothesis H_{01}. Therefore, we conclude that our approach has better performance (for APIs with low abundance) than the baseline approach that uses frequent-sequence mining.

We can also see from Fig. 7 that the F-measure of our approach is higher than that of the baseline in mining usage patterns for APIs with high or low dominance. The improvement of F-measure for the APIs with two different dominance levels averages 9% and 24% respectively. In addition, as shown in Table 5, the p-values of the performance comparison in mining usage patterns for API with high or low dominance are significant, so we can reject the null hypothesis H_{02}. Therefore, we conclude that our approach performs better in recommending usage patterns for APIs with high dominance comparing to the baseline approach that uses frequent-sequence mining. In addition, based on the Cliff's delta values shown in Table 5, we can see the F-measure values of the baseline approach and our approach for APIs with low abundance or high dominance have few overlaps.

More than half of the F-measure values of UP-Miner are worse than that of our approach. Therefore, we can conclude that our approach significantly outperforms the frequent-sequence-mining based approach for queries related to imbalanced API usages with a large effect size. This is due to the fact that usage patterns of such less frequently used APIs or the non-dominant usage patterns of some APIs are more likely to be ignored by a frequent-sequence-mining based approach.

We further analyze the experimental result using the example API query `BluetoothSocket.connect()`. Based on the identified representative object usages, O_1(Fig. 5), O_4 (Fig. 5), O_5(Fig. 5) and O_9(Fig. 5), for the API query `BluetoothSocket.connect()`, usage patterns P_a (Fig. 1), P_b (Fig. 1), P_c (Fig. 1) and another usage pattern containing both `BluetoothSocket.getInputStream()` and `BluetoothSocket.getOutputStream()`, are recommended by our approach. Usage patterns P_b (Fig. 1), P_c (Fig. 1), and P_r (Fig. 2) are recommended by UP-Miner. We observe that our approach recommends more complete API usage patterns comparing to UP-Miner. The code snippets corresponding to usage patterns P_a and P_r are `GetInformation()`(Fig. 1) and `run()`(Fig. 8). We can see from Table 6 that all the participating methods of expected API usage patterns for query `BluetoothSocket.connect()`have been contained in the usage pattern P_a recommended from our approach while three methods of API `BluetoothDevice` are missing in the usage pattern P_r recommended from Codota. The missing method, `createRfcommSocketToServiceRecord()`,is necessary since it is the method that creates a Bluetooth- Socket ready to start a connection to a remote device.

Table 6
Summary of methods contained in API usage patterns P_a and P_r recommended for query BluetoothSocket.connect() from our approach and Codota.

Participating methods of expected usage pattern	Usage pattern P_a from our approach contains or not	Usage pattern P_r from UP-Miner contains or not
BluetoothAdapter. cancelDiscovery()	√	√
BluetoothDevice. createRfcommSocketToServiceRecord()	√	X
BluetoothSocket. connect()	√	√
BluetoothDevice. getName()	√	X
BluetoothDevice. getAddress()	√	X
BluetoothSocket. close()	√	√

```
57  public void run() {
58      Log.i(TAG, "BEGIN mConnectThread SocketType:" + mSocketType);
59      setName("ConnectThread" + mSocketType);
60
61      // Always cancel discovery because it will slow down a connection
62      mAdapter.cancelDiscovery();
63
64      // Make a connection to the BluetoothSocket
65      try {
66          // This is a blocking call and will only return on a
67          // successful connection or an exception
68          mSocket.connect();
69      } catch (IOException e) {
70          // Close the socket
71          try {
72              mSocket.close();
73
74          } catch (IOException e2) {
75              Log.e(TAG, "unable to close() " + mSocketType +
76                      " socket during connection failure", e2);
77          }
78          if (listener != null) listener.connectionFailed(mDevice);
79          return;
80      }
```

Fig. 8. The code snippet corresponding to the usage pattern $P_r$returned by UP-Miner for API query: BluetoothSocket.connect().

Algorithm 1 Detecting communities in an object usage network.

Input: The object usage network G(V, E)
Output: The community membership of object usages V
Initial state: Each object usage represents one community
calculate the initial value of modularity using Eq. (1)
for each object usage i in V do
  for each object usage j in V **do**
    compute ΔM_{ij} using Eq. (2)
  end loop
end loop
select the maximum ΔM_{ij}^{max}
while $\Delta M_{ij}^{max} > 0$ do
  join corresponding communities C_i and C_j
  increment M by ΔM_{ij}^{max}
  update the matrix ΔM based on joining activity
  select new ΔM_{ij}^{max} from updated ΔM matrix
end loop

Although the missing methods might occur in a code snippet in the repository, UP-Miner extracts an API usage pattern based on the frequent-itemset mining which considers popularity, therefore, the final answer recommended by UP-Miner is incomplete. The advantage of our approach is due to the fact that we recommend API usage patterns based on the co-existence relations between object usages instead of only the popularity of API usage patterns.

Our approach can achieve better performance than the frequent-sequence-mining based approach in answering queries related to APIs with imbalanced usage.

RQ 3: Can our approach effectively rank API usage patterns?

Motivation. It is challenging to properly rank the mined API usage patterns for a query (Bruch et al., 2008). A successful ranking should place the expected API usage pattern at the top of the result list before the other API usage patterns. A proper ranking of the mined usage patterns can save developers' time since developers do not need to review a large amount of suggested answers before finding the expected answer within the ranked result list (Wang et al., 2013). Hence, we evaluate the performance of our approach in ranking the mined usage patterns.

Approach. We adopt the automated evaluation process (Bruch et al., 2008) described in Section 4. B. 2) to evaluate our approach. The evaluation uses a set of 100 randomly selected queries and the corresponding expected answers generated by the automated process. We compare the ranking performance of our approach with our baseline, Codota, which is an online recommendation engine with ranking capability. We execute the queries on Codota and our approach. We study the quality of ranking using the Normalized Discounted Cumulative Gain (NDCG) measure (Manning et al., 2008). NDCG measures whether highly relevant answers are placed at the top of the result set. NDCG is defined in Eq. (8) where Discounted Cumulative Gain (DCG) is computed using Eq. (7). Ideal DCG ($IDCG_p$) is the maximum possible DCG, which can be computed following Eq. (7) by sorting the API usage patterns in the result set based on the relevance to the expected answer. We use Jaccard coefficient (Jaccard, 1901) to compute the relevancy used in NDCG definition as $rel_i = \frac{P_e \cap P_i}{P_e \cup P_i}$, which compares the similarity between the methods in expected answer, P_e, and the i^{th} mined usage pattern, P_i.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{[rel_i+1]} - 1}{\log_2(i+1)} \tag{7}$$

$$NDCG_p = \frac{DCG_p}{IDCG_p} \tag{8}$$

where rel_i *is the relevance of the result at position i to the expected answer;* p *is the total number of the result list.*

We apply Wilcoxon rank sum test (Sheskin, 2007) with 5% confidence level (*i.e.*, p-value < 0.05) and compute Cliff's delta to evaluate the ranking quality differences of our approach and Codota.

Results. The result of NDCG is summarized in Fig. 9. The ranking quality of our approach is better than Codota with an average of 3.59% improvement onNDCG, from 87.37% in Codota to 90.96% in our approach. 61% queries have better NDCG values from our approach than Codota. Based on the p-values (0.01), we can conclude that our approach can significantly outperform Codota in terms of ranking quality.

Our approach can effectively rank API usage patterns by 3.59% improvement on NDCG comparing to Codota.

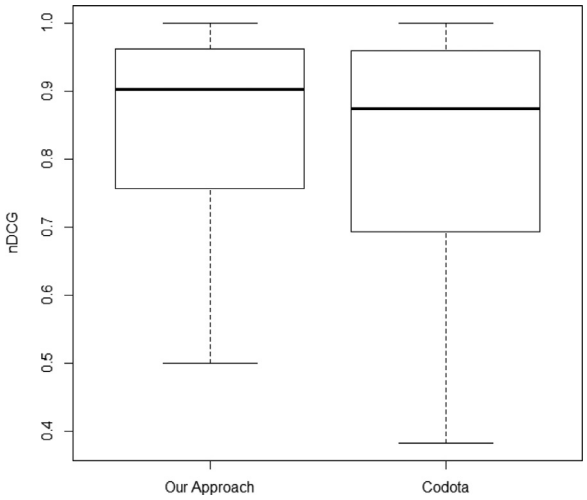


Fig. 9. NDCG distribution comparison.

5.2. Preliminary user study

We conducted a small survey of three developers from the industry to investigate whether the API usage patterns recommended by our approach can help their programming tasks. All the three developers have more than five-year programming experience. Before participating the survey, they all received a 30-minutes' training on the usage of API usage patterns and the general idea of our code recommendation approach. First, we randomly selected 10 API queries from the ones used for RQ1 and RQ2. Then, we asked developers to independently examine the API usage patterns recommended for the 10 API queries in terms of completeness and duplication and give scores on the usefulness of the recommended answers. The scores are given using typical five-level Likert items (strongly agree - 5, agree - 4, neutral - 3, disagree - 2, and strongly disagree - 1). The average scores for the 10 queries examined by the three developers are 4.3, 4.1 and 3.9, respectively.

We also conducted an initial interview with the three developers and asked them to provide detailed feedback on our API usage pattern recommendation. We asked the following questions during the interview.

(1) Is the API usage pattern recommendation approach useful to your programming tasks?

If the answer to question (1) is yes, then answer the questions (2)–(5).

(1) Why is the approach useful?
(2) How is the approach useful?
(3) In what circumstance is the approach most useful?
(4) Any suggestions for the further improvement on the approach?

If the answer to question (1) is no, then answer the questions (6)–(7).

(1) Why do you think the approach is not useful?
(2) How should the approach be further improved?

All of the three developers gave positive comments on the usefulness of our approach. We have summarized their answers to the interview questions. The detailed feedback is shown as follows.

a. **Why is the approach useful?**
(1) The proposed approach is complementary to the existing API documentation to understand API usage.
(2) The approach can help the developers effectively and efficiently (re)use APIs to improve development productivity and quality.

b. **How is the approach useful?**
(1) The approach can place the desired usage patterns at the top of the ranked result list so that developers can easily find it.
(2) The usage pattern recommendation approach can provide code examples for each recommended usage pattern. The code examples show the detailed context of the API usage. It helps the developers easily understand the API usages.
(3) The complete API usage pattern recommendations show how an API method should be used in different contexts to complete different programming tasks.

c. **In what circumstances is the approach most useful?**
(1) The approach is really helpful in learning how the API should be used, especially for some less commonly-used APIs.
(2) The approach can recommend more complete API usage patterns for frequently-used APIs.
(3) The redundancy of the API usage patterns recommended for the frequently-used APIs is low.

d. **How can the approach be improved further?**
(1) The approach should be further evaluated by applying it in recommending API usage patterns for different programming languages.
(2) The approach used to reduce the similarity of identified API usage patterns is not intuitive, and more techniques could be further explored to make the approach easier to adopt.
(3) The approach can only recommend API usage patterns used in a single method body. In the future, recommending API usage patterns used across different methods might need to be studied.

Based on the results of the small-scale, preliminary user study, we can draw an initial conclusion that the API usage patterns recommended by our approach can help developers with their programming tasks.

6. Threats to validity

In this section, we analyze the threats to validity of our study following the guidelines provided by Robert (2002).

Threats to construct validity concern whether the setup and measurement in the study reflect real-world situations. For our study, the construct validity threats mainly come from the establishment of the "gold set" for API queries. We build the "gold set" manually based on our programming knowledge. The subjective knowledge about the Android programming may affect the establishment of the "gold set", which will further affect the values of evaluation metrics. To reduce the subjectivity of build the "gold set", we use the same setup as earlier studies (Wang et al., 2013; Thummalapenta and Xie, 2007) on API usage pattern mining by including three curators having one-year Android application development experience or more than five years of Java programming experience. Furthermore, whenever it is possible, *i.e.*, RQ3, we use a completely automatic approach to evaluate the performance to reduce the subjectivity.

Threats to internal validity concern the uncontrolled factors that may affect the experiment result. In our experiment, the main threat to internal validity is the implementation of two clustering algorithms. We use a greedy algorithm to detect communities in a network of object usages as a practical solution to handle our large-scale data. For hierarchical clustering, we use Gamma quality

index to determine the best number of clusters. Gamma quality index can be used to generate the clusters with the maximum distance between any pairs of object usages from different clusters. The result can be varied if we use different methods in our work.

Threats to conclusion validity concern the relation between the treatment and the outcome. We have used a non-parametric test that does not hold assumption on the distribution of data. It is possible that some of expected API usage patterns cannot be found in the corpus of Codota. In our study, we only consider the queries supported by Codota.

Threats to external validity concern whether our experimental results can be generalized for other situations. The main threat to external validity in our study is the representativeness of our API queries. In our experiment (RQ1, 2, and 3), we used 120 queries in total, which covers different aspects of API usage, such as database operation, socket communication, cookie management. The size of our query set is comparable to similar studies on code search and recommendation, *e.g.,* (Wang et al., 2013; Thummalapenta and Xie, 2007).

Threats to reliability validity concern the possibility of replicating this study. We provide the necessary details needed to replicate our work. Replication package is publicly available (Replication package).

7. Related work

In recent years, researchers have conducted various studies (Wang et al., 2011; McMillan et al., 2012; Reiss, 2009; Holmes and Murphy, 2005) on code search and recommendation. Such studies aim to find the relevant code snippets for a given query. In our study, we focus on a different problem of code search which recommends API usage patterns. In this section, we review the studies related to mining API usage patterns.

Most of the existing usage pattern mining approaches use the frequent-sequence mining technique to identify usage patterns. MAPO (Zhong et al., 2009) developed by Xie and Pei has established a framework for mining API usage patterns automatically from open source repositories. It uses existing code search engines and the Bitmap algorithm to mine frequent-sequences. MAPO can produce a large number of redundant sequences (Wang et al., 2013). Nguyen et al., (2009) introduce GrouMiner, a graph-based approach for mining usage patterns of multiple objects. GrouMiner represents object usages in a scenario as a directed graph and then detects the (sub)graph that frequently appears in the set of object usage graphs. Acharya et al., (2007) propose a framework to extract API usage-scenarios from control-flow-sensitive static traces related to the APIs of interest and then summarize different usage scenarios as frequent partial orders to be considered as the API patterns. The aforementioned work uses frequent-sequence mining techniques which may overlook infrequent usage patterns and therefore affects the completeness of the result set. Our approach does not rely on frequent-sequence mining to identify API usage patterns.

To solve the problem of the frequent-sequence mining technique, existing studies, such as (Dong et al., 2007; Yun et al., 2003; Szathmary et al., 2010; Hipp et al., 2000), propose approaches to mine infrequent itemsets. Dong et al., (2007) aim to get more valued negative association rules by discovering infrequent itemsets. They assign various minimum supports to itemsets with different lengths in order to discover both infrequent itemsets and frequent itemsets. Similarly, to mine association rules from significant rare data, Yun et al., (2003) use relative support values instead of defined support values in the process of itemset mining. Szathmary et al., (2010) use an algorithm called Apriori-Rare to retain rare itemsets instead of pruning them. However, it would generate explosive number of output itemsets. Hipp et al., (2000) introduce two approaches to determine support values in the process of itemset mining. Different from the existing approaches (Dong et al., 2007; Yun et al., 2003; Szathmary et al., 2010; Hipp et al., 2000), our approach is a graph-based approach instead of an itemset mining-based approach which requires a support value as the essential configuration. It is hard to determine the support value in practices. Our approach does not need to tune the support value.

The existing studies (Dong et al., 2007; Yun et al., 2003; Szathmary et al., 2010; Hipp et al., 2000) attempt to include a support value to solve the problem of the frequent-sequences mining technique, but they have not been applied in the area of code search. Up-Miner (Wang et al., 2013) is one of the representatives of the existing techniques, which addresses the problem of the frequent-sequences mining technique in the area of code search. UP-Miner applies BIDE (Wang and Han, 2004) algorithm which uses the frequent-sequence mining technique to search for popular call sequences. UP-Miner includes a two-step clustering strategy to reduce the redundancy and increase the coverage of the returned usage patterns. We compare the performance of our approach with Up-Miner.

Moritz et al., (2013) implement an interactive code search tool which does not use the itemset mining-based approach. It requires the developers to continuously select the API methods related to specific tasks. Our approach only requires developers to specify an initial API query, and then returns different usage patterns for the query.

8. Conclusion

In this paper, we have proposed an approach to mine and recommend API usage patterns. We extract the co-existence relations among object usages within same methods in our corpus and model the object usages and their relations using a network. Then, we apply two clustering techniques to balance the tradeoff between the completeness and redundancy on the recommended API usage patterns for a query. We conducted an empirical study using a large-scale corpus consisting of 11,510 Android applications. Our case study shows that our approach outperforms the state of the art frequent-sequences-mining based approaches. Our approach can mine API usage patterns that achieve high completeness and low redundancy. Specifically, we show that our approach outperforms the baseline in mining less frequently used API usage patterns. In addition, the ranking quality of our approach is better than Codota which is an online commercial usage pattern recommendation service for Android development.

The API usage patterns recommended by our approach can support developers in programming with unfamiliar APIs, especially the APIs that are experiencing imbalance usage (Mendez et al., 2013). Since the clustering techniques used in our approach is not specific to Android dataset, in the future, we plan to replicate our study to recommend API usage patterns of different programming languages using other open-source software projects. We will explore other techniques, such as (Yun et al., 2003; Szathmary et al., 2010) that have been applied to address the infrequent-itemset mining problem in data mining area, and apply them in the code search area. Moreover, we plan to conduct a more comprehensive user study by inviting a larger developers to participate in the user study.

Acknowledgment

We thank Mr. Hammad Khalid, Dr. Meiyappan Nagappan and Dr. Ahmed E. Hassan at Queen's University for providing us the raw data used in our case study.

References

Acharya, M., Xie, T., Pei, J., Xu, J., 2007. Mining API patterns as partial orders from source code: from usage scenarios to specification. In: Proceedings of 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 25–34.

APKTool. http://code.google.com/p/android-apktool/.

Bajracharya, S., Ngo, T., Linstead, E., Rigor, P., Dou, Y., Baldi, P., Lopes, C., 2006. Sourcerer: a search engine for open source code supporting structure-based search. In: Proceedings of International Conference on Objected-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 681–682.

Bruch, M., Schäfer, T., Mezini, M., 2008. On evaluating recommender systems for API usages. In: Proceedings of International Workshop on Recommendation Systems for Software Engineering, pp. 16–20.

Buse, R.P.L., Weimer, W., 2012. Synthesizing API usage examples. In: Proceedings of 34th International Conference on Software Engineering (ICSE), pp. 782–792.

Clauset, A., Newman, M.E.J., Moore, C., 2004. Finding community structure in very large networks. Phys. Rev. E 70, 066111.

Cliff, N., 1993. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. Psychological Bulletin.

Codota. An onlinde usage pattern recommendation engine for Andriod application developmen, www.codota.com/.

Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences, 2nd edn. Lawrence Erlbaum.

Cohen, J., 1992. A Power Primer. Psychological Bulletin.

Dong, X., Niu, Z., Shi, X., Zhang, X., Zhu, D., 2007. Mining both positive and negative association rules from frequent and infrequent itemsets. In: Advanced Data Mining and Applications. Springer, Berlin, Heidelberg, pp. 122–133.

Grahne, G., Zhu, J., 2003. Efficiently using prefix-trees in mining frequent itemsets. In: Proceedings of IEEE ICDM Workshop on Frequent Itemset Mining Implementations.

Guerra, L., Robles, V., Bielza, C., Larrañaga, P., 2012. A comparison of clustering quality indices using outliers and noise. Intell. Data Anal. 16 (4), 703–715.

Han, J., Kamber, M., 2006. Data Mining: Concept and Techniques, 3rd ed. Elsevier Press.

Hipp, J., Güntzer, U., Nakhaeizadeh, G., 2000. Algorithms for association rule mining—a general survey and comparison. ACM SIGKDD Explor. Newsl. 2 (1), 58–64.

Hoffmann, J., Ussath, M., Holz, T., Spreitzenbarth, M., 2013. Slicing droids: program slicing for smali code. In: Proceedings of 28th Symposium On Applied Computing (SAC), pp. 1844–1851.

Holmes, R., Murphy, G.C., 2005. Using structural context to recommend source code examples. In: Proceedings of 27th International Conference on Software Engineering (ICSE), pp. 117–125.

Jaccard, P., 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin de la Société Vaudoise des Sciences Naturelles 37, 547–579.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Penta, M.D., Oliveto, R., Poshyvanyk, D., 2013. API change and fault proneness: a thread to the success of Android apps. In: Proceedings of 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 477–487.

Manning, C.D., Raghavan, P., Schütze, H., 2008. Introduction to Information Retrieval. Cambridge University Press.

McMillan, C., Hariri, N., Poshyvanyk, D., Cleland-Huang, J., Mobasher, B., 2012. Recommending source code for use in rapid software prototypes. In: Proceedings of 34th International Conference on Software Engineering (ICSE), pp. 848–858.

Mendez, D., Baudry, B., Monperrus, M., 2013. Empirical evidence of large-scale diversity in API usage of objected-oriented software. In: Proceedings of 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 43–52.

Mishne, A., Shoham, S., Yahav, E., 2012. Typestate-based semantic code search over partial programs. In: Proceddings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 997–1016.

Mojica, I.J., Adams, B., Nagappan, M., Dienst, S., Berger, T., Hassan, A.E., 2014. A large scale empirical study on software reuse in mobile apps. In: Proceedings of IEEE Software, pp. 78–86.

Monperrus, M., Mezini, M., 2012. Detecting missing method calls as violations of the majority rule. ACM Trans. Softw. Eng. Methodol. 22 (1).

Moritz, E., Vásquez, M.L., Poshyvanyk, D., Grechanik, M., McMillan, C., Gethers, M., 2013. ExPort: detecting and visualizing API usages in large source code repositories. In: Proceedings of International Conference on Automated Software Engineering, pp. 646–651.

Newman, M.E.J., 2004. Fast algorithm for detecting community structure in networks. Phys. Rev. E 69, 066133.

Newman, M.E.J., Girvan, M., 2004. Finding and evaluating community structure in networks. Phys. Rev. E 69, 026113.

Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N., 2009. Graph-based mining of multiple object usage patterns. In: Proceedings of 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 383–392.

Reiss, S.P., 2009. Semantics-based code search. In: Proceedings of International Conference on Software Engineering, pp. 243–253.

Replication package. https://www.dropbox.com/s/3chjl0fpauezphp/usage%20pattern%20Replication.rar?dl=0.

Robert, K.Y., 2002. Design and Methods, 3rd ed. International Educational and Professional Publisher.

Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J., 2006. Appropriate statistics for ordinal level data: should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In: AIR Forum, pp. 1–33.

Scaffidi, C., 2005. Why are APIs difficult to learn and use? Crossroads 12 (4) 4--4.

Sheskin, D.J., 2007. Handbook of Parametric and Nonparametric Statistical Procedures, 4th ed. Chapman and Hall/CRC.

Szathmary, L., Valtchev, P., Napoli, A., 2010. Finding minimal rare itemsets and rare association rules. In: Knowledge Science, Engineering and Management. Springer, Berlin, Heidelberg, pp. 16–27.

Thummalapenta, S., Xie, T., 2007. Parseweb: a programmer assistant for reusing open source code on the web. In: Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 204–213.

Vásquez, M.L., Holtzhauer, A., Cárdenas, C.B., Poshyvanyk, D., 2014. Revisiting Android reuse studies in the context of code obfuscation and library usages. In: Proceedings of Working Conference on Mining Software Repositories (MSR), pp. 242–251.

VisionMobile, 2013. Developer tools: The Foundations of the App Economy. Developer Economics.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D., 2013. Mining succinct and high-coverage API usage patterns from source code. In: Proceedings of 10th Working Conference on Mining Software Repositories (MSR), pp. 319–328.

Wang, J., Han, J., 2004. BIDE: efficient mining of frequent closed sequences. In: Proceedings of 20th International Conference on Data Engineering (ICDE), pp. 79–90.

Wang, S., Lo, D., Jiang, L., 2011. Code search via topic-enriched dependence graph matching. In: Proceedings of 18th Working Conference on Reverse Engineering (WCRE), pp. 119–123.

Yun, H., Hab, D., Hwanga, B., Ryuc, K.H., 2003. Mining association rules on significant rare data using relative support. J. Syst. Softw. 181–191.

Zhong, H., Xie, T., Pei, P., Mei, H., 2009. MAPO: mining and recommending API usage pattern. In: Proceedings of European Conference on Object-Oriented Programming, pp. 318–343.