applied sciences

MDPI

Article

Mining Hidden and Fragmented API Usages in Android Applications

Mingwan Kim and Neunghoe Kim *

Department of Computer Science and Engineering, Korea University, Seoul 02841, Korea; tolskais@korea.ac.kr
* Correspondence: nunghoi@korea.ac.kr

Received: 25 October 2020; Accepted: 15 December 2020; Published: 17 December 2020

Abstract: Application Programming Interface (API) usage mining is an approach used to extract the common API usage to help developers get used to the APIs. However, in Android applications, the usage can be hidden or fragmented due to class inheritance. Such hidden or fragmented usages could decrease the coverage and accuracy of the existing API mining approaches. Our method aims to resolve the problem of hidden and fragmented usages through API generalization. This generalized usage is expected to be applicable to every class that inherits a class in the usage. In the experiment, among 442,809 Android API usages, 104,839 usages either were hidden or fragmented. By revealing such usages, the accuracy of the code completion was improved by at most 6.66%. The usage generalization was efficient for extracting API usages in Android applications in which the APIs are used through class inheritance.

Keywords: API usage patterns; Android applications; object-oriented programming

1. Introduction

Recently, software development has relied on third-party libraries instead of writing the code from scratch. These libraries define their own Application Programming Interfaces (APIs) that provide various functionalities required for developing software programs. However, it has been difficult for developers to grow accustomed to these APIs [1]. To resolve this, existing libraries have provided developers with several materials such as API documentation, unified name convention rule, and example code. However, these approaches are insufficient to present various API usages [1].

API usage mining is another approach that aims to resolve the difficulties in learning APIs. Existing mining approaches, such as association rule [2], graph representation [3], and clustering [4], have been proposed to extract common API usages from huge source repositories. Such common usages can be used for various purposes, such as code completion [5] and document generation [6].

This paper focuses on the hidden and fragmented usages when the API usages are collected in Android applications. Assume that a developer-written class X inherits an Android library class Activity. Such a class X can invoke an API method getApplicationContext() (abbreviated as gAC()), which aims to manipulate the global application status. Without considering the class inheritance, an API mining tool would ignore the call site X.gAC(), which is a non-API method defined in X. In other words, the usages involving X.gAC() will be fragmented. In the worst case, if such usages consisted of the method invocations only with X, they would be hidden from the mining tool.

The fragmented and hidden usages should be handled properly to increase the accuracy and the coverage of an API usage mining tool. These two metrics have been used to evaluate the performance of existing mining tools [4,5]. An API mining tool would infer an incorrect API usage from the fragmented usage as the tool misses the inherited APIs. In the similar manner, the API mining tools could not detect the hidden usages, which would decrease the coverage of the mining tool.

In this paper, we propose a generalization approach to minimize the number of fragmented or hidden API usages. This approach aims to generalize the API usages by finding the representative classes for each usage. An evaluation was conducted using 442,809 API usages demonstrated that 104,839 usages (23%) were hidden or fragmented without the generalization. In addition, the accuracy of the code completion increased by at most 6.66% with the generalized usages. These experimental results imply the importance of usage generalization when mining API usages.

The remainder of this paper is structured as follows. Section 2 introduces the existing studies related to our work. Section 3 presents an example code that motivates our idea. Section 4 explains the approach for generalizing the API usages to minimize the fragmented or hidden usages. Section 5 presents the implementation of the code completion with the proposed approach. The experimental setup and results are explained in Sections 6 and 7, respectively. Section 8 discussed with the experimental results. Finally, Section 9 summarizes the proposed approach and the main findings.

2. Related Work

In recent years, the API mining tools have been advanced in various aspects [7]. In particular, these tools have focused on collecting high-quality usages from huge source repositories. This section introduces several existing relevant studies.

2.1. Existing API Usage Mining Tools

Mining API usages from Open-source repositories (MAPO) [4,8] is a tool proposed for providing frequent API usages for a given API query. To search the common usages, the API sequences are clustered based on three similarities computed from the method names, API members, and subsequences in each cluster.

Usage Pattern Miner (UP-Miner) [9] was proposed to improve the quality of API usages by removing the redundant ones among them. This goal was achieved by mining frequent closed sequences based on SeqSim, which is an n-gram based similarity metric. For example, given the API sequences a, b, and ab, UP-Miner attempts to select the longest sequences ab. Despite the improvement, the metric could not reflect the similarity between the usages in which the same library APIs involved through class inheritance.

Nguyen et al. [10] proposed a graph representation of the API usages involving multiple objects. Based on the graph representation, common API usages could be found by finding the frequent induced subgraphs. It was demonstrated that this approach was effective in detecting anomalies that could cause the defects in a program.

Method USage Examples (MUSE) [11] provided example codes extracted from the source code with the clone detection technique. The clone detection technique was used to provide one representative usage among the clones.

Probabilistic API Miner (PAM) [5] aims to overcome the limitation that the frequency-based mining approach tends to include uninteresting usages. The basic idea is to include only the API sequence that improves the overall probability of the API model.

API FunctiOn Calls and USage patterns (FOCUS) [12] is a state-of-the-art recommendation system based on the collaborative filtering. The basic idea is that the valuable API usages in a target program can be collected from the similar program. The experimental results demonstrated that FOCUS outperformed PAM.

LibraryGuru [13] focused on the API usages found in event-based frameworks such as Android. This tool recommends the most suitable callback methods for a given API query.

All the aforementioned tools focused on mining frequent API usages while reducing redundant usages. Although these tools contribute to improving the collected dataset, several API usages could be either fragmented or hidden because of the APIs used by inheriting library classes.

2.2. API Usage Mining with the Class Inheritance Information

CodeWeb [2] proposed generalized association rules to consider class inheritance for extracting API usage patterns. This work is similar to our study in that class inheritance was considered when extracting API usages from the source code. However, this approach did not discuss how the rules can be combined with recent learning models such as the Hidden Markov Model (HMM), *n*-gram, or recurrent neural networks (RNN). Our study aims to consider the class inheritance for such various models, especially for HMMs.

Pattern-based Bayesian Networks (PBN) [14] is a code completion tool based on the Bayesian network. In contrast to the other studies introduced in Section 2.1, the class hierarchy was considered to determine (1) the enclosing method through which an API sequence is extracted, and (2) resolving the overloading methods. In contrast to these purposes, our approach considers the class inheritance for each method call in an API sequence to reduce the fragmented or hidden sequences.

Robbes and Lanza [15] compared several strategies that could be used for code completion. The experiments demonstrated that the strategy that prioritizes the APIs changed recently was more effective than the other strategies such as prioritizing the APIs found in the parent classes. However, this comparison did not consider that the class hierarchy could be utilized to resolve problematic usages.

Bruch et al. [16] proposed a document generation method for the object-oriented white-box frameworks. The document is generated to introduce how each library class should be inherited. Although this approach focuses on class inheritance, the fragmented or hidden usages are ignored.

2.3. Data Source and Learning Model of the API Usages

It is possible to collect API usages from the binary or source files. For example, hidden Markov model of API usages (HAPI) is a statistical and generative model based on the dataset collected from binary files [17]. In contrast, Non Client-Based Usage Patterns miner (NCBUP-miner) attempted to extract usage patterns from the framework code, instead of the client code [18]. In our study, we collected the usages from the source code based on the static code analysis.

Ogasawara et al. [19] focused on the characteristics of API usages that influenced existing API-related systems. The investigation confirmed that frequent API usages have been observed in various projects. In contrast, it also confirmed that the uncommon API usages should not be ignored as they could be project-specific API usages.

The mined API usages have been learned using various prediction and representation models. Raychev et al. [20] considered the code completion task as a natural-language processing (NLP) task. Based on this idea, a language model was implemented based on *n*-gram and recurrent neural networks (RNNs) that have been used in the NLP domain. Furthermore, the *n*-gram model was improved by combining syntax and semantic information [21]. an *n*-gram was also used to extract the code convention in the source code [22]. Similarity, Yan et al. [23] proposed an API suggestion model based on long short-term memory (LSTM).

Graph-based statistical Language (GraLan) [3] learns API usages based on the probability that each available graph contains the given subgraph. Maddison et al. [24] proposed the probabilistic context-free grammar (PCFG) and demonstrated that the source code generated by the model was human-friendly source code. CodeKernel [25] was proposed to increase the quality of the usage dataset. It was demonstrated that the object usage graph and the graph kernel were effective in increasing the dataset quality.

Among these models, we adopted HAPI [17] that showed better accuracy for code completion and has a relatively feasible learning time than the other models that are based on *n*-gram and RNN. In addition, this model has been used in a recent study that proposed a code search engine using natural languages [26].

2.4. Applications of the API Usages

Mined API usages and the prediction models have been used for various purposes. DroidAssist [27] detected suspicious API usages in the source code based on HMM. Java Rule Finder (JRF) [28] aims to extract the specification of a library from the source code written in object-oriented languages. The program rule graph (PRG) was proposed to represent the pre- and post-conditions of each method. Schäfer et al. [29] proposed a method for extracting the changes in API usages after the framework code was modified. Gu et al. [30] proposed a search engine based on a deep learning model to find the API usages. Given a query in natural language, their proposed system provides a relevant example code. Export [31] is a visualization tool that presents the API usages similar to the given usage. Such visualization aims to help developers investigate API usages in huge source repositories. Jigsaw [32] is a model that integrates the reused source code into a new location. Nguyen et al. [33] proposed the method that extracted API usages from fine-grained changes.

The proposed API generalization is evaluated for the code completion task. For example, GraPacc [34] is a graph-based API mining tool utilized for performing code completion using context-sensitive features collected from code editing. Moreover, this task has been used in numerous existing studies, including those studies introduced in Sections 2.1 and 2.2.

3. Motivation

Table 1 presents the two API usages collected from the projects wordpress-android and Pockethub. Each method invocation is presented as a qualified name that consists of the names of the class and method. One usage consists of the two classes, WPLaunchActivity and Intent, while another usage consists of LoginActivity and Intent. These two raw API usages of WPLaunchActivity and LoginActivity seem incompatible. However, both usages aim to run a new activity and finish the current activity. This similarity can be identified through API generalization. The two generalized usages consist of the methods bound to the classes, Intent and Activity. These classes can be determined by traversing their class hierarchies, as shown in Figure 1.

Table 1. Example of raw and generalized Application Programming Interface (API) sequences.

Class Name	Type	API Sequence
WPLaunchActivity	Raw	Intent.<init> WPLaunchActivity.startActivity WPLaunchActivity.finish
	Generalized	Intent.<init> Activity.startActivity Activity.finish
LoginActivity	Raw	Intent.<init> LoginActivity.startActivity LoginActivity.finish
	Generalized	Intent.<init> Activity.startActivity Activity.finish

The raw usages are generalized by finding the representative class that defines all the methods that appear in each usage. In this example, LoginActivity has five candidate classes, one of which can be its presentative class. In particular, the ancestor classes Context and Activity define startActivity() and finish(), respectively, which are shown in the raw usage of LoginActivity. The API generation determines that Activity would be a representative class as Context could not invoke finish() defined in Activity. Similarly, WPLaunchActivity can also be generalized into Activity.

Consequently, the two raw usages are converted into the same usage that initializes Intent and invokes startActivity() and finish(), which are bound with Activity.

Existing studies have focused on reducing the redundant usages to increase the quality of the usage dataset [12,25]. However, we will demonstrate that frequent usages could be missed, or several API calls could be omitted when collecting API usages from the raw sequences.

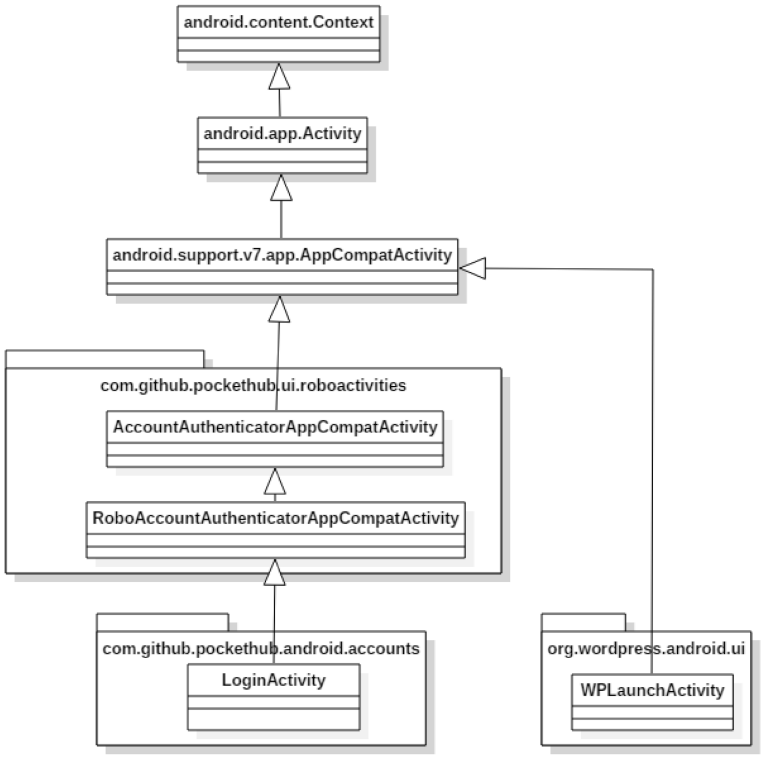

Figure 1. Class hierarchy of WPLaunchActivity and LoginActivity.

4. API Usage Generalization

Let $\boldsymbol{T} = \{t_1, \ldots, t_m\}$ be a set of types and $\boldsymbol{M} = \{m_1, \ldots, m_p\}$ be a set of method signatures. An API usage is a sequence of the pairs (t_i, m_j), or $\boldsymbol{S} = \{(t, m) \mid t \in \boldsymbol{T} \text{ and } m \in \boldsymbol{M}\}$. We assume that API usages are collected from a compliable and error-free code. For example, the method add() declared in java.util.List can be paired, whereas the method cannot be paired with java.util.Map.

Figure 2 shows the algorithm used to generalize the input API sequence $\boldsymbol{S}$. This algorithm iterates each API sequence twice. The first iteration collects the class that defines each method in the given $\boldsymbol{S}$. This iteration targets only the pair (t, m) that invokes an overridable method (line 3). For example, in Java, every method can be overridden unless the method is a static or private method [35]. Subsequently, the algorithm finds the method to be invoked from the pair (line 4). The notation $resolution(t, m)$ denotes the language-specific method resolution (e.g., [36]). Thereafter, the identified class t_d replaces the original class t (line 5). Lastly, the identified class is added to the variable classes (line 6).

After the first iteration, the algorithm determines the representative classes (lines 8–13). The variable called classes consists of classes that are partially ordered by class inheritance. In other words, each minimal element in classes should have a parent class and zero child class. Every non-minimal class is mapped into the corresponding minimal class (lines 9–13).

The second iteration aims to replace each class with a representative class in the API sequence. For each pair (t, m), t is replaced as the corresponding minimal element identified in the variable *mapping* (line 15–19).

Figure 3 illustrates the generation process with the code that instantiates the class CategoryFragment. The first iteration investigates each pair in the usage to determine the class that declares the method to be invoked from the pair. As shown, the first pair (Bundle, <init>) and the

third pair (CategoryFragment, <init>) are not modified as these pair aim to invoke non-overridable methods; <init> is a special method used to initialize a class. In contrast, in the second pair (Bundle, putString), Bundle is replaced with BaseBundle, which defines the method to be invoked from this pair. Similarity, in the pair (CategoryFragment, setArgument), CategoryFragment is replaced with Fragment. In the second iteration, the class of each pair is replaced as the representative class in this API usage. In this example, there is only one candidate for each class: BaseBundle for Bundle and Fragment for CategoryFragment.

function GeneralizeSequence (Sequence S)

1 $classes = \emptyset$
2 for each (t, m) in S {
3 if (t, m) invokes a overridable method {
4 $(t_d, m_d) = resolution(t, m)$
5 replace t as t_d and insert t_d into $classes$
7 }
8 $mapping = \emptyset$
9 for each minimal element t_c in $classes$ {
10 $C = \{t \mid t$ is a parent class of $t_c\}$
11 for each t in C {
12 insert (t, t_c) into $mapping$
13 }
14 }
15 for each (t, m) in S {
16 if (t, m) invokes a overridable method {
17 replace t to t_c where (t, t_c) in $mapping$
18 }
19 }
20 return S

Figure 2. API sequence generation algorithm.

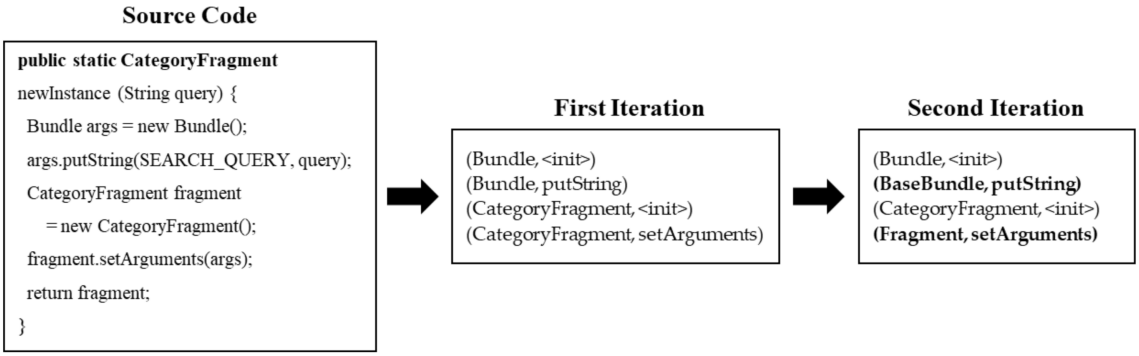

Figure 3. Example of the API generation process.

5. Code Completion with the Generalized API Sequence

This section introduces the code completion process for Android projects with the proposed generalization method, as illustrated in Figure 4. First, the training phase generalizes the API sequences collected from the source code. Subsequently, statistical models were generated based on these generalized sequences. Later, in the test phase, code completion was conducted using these generated models. The details of each step are described in the following subsections.

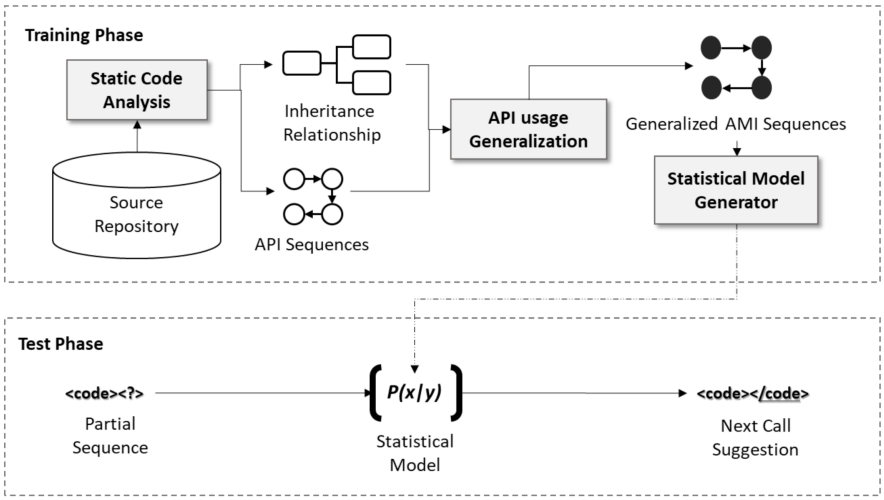

Figure 4. Overview of the code completion with the API sequence generation.

5.1. API Usages Dataset

The API usages can be collected either from the source code or the compiled bytecode. A larger usage dataset can be collected from the compiled bytecode than from the source code [17]. For example, API usages are difficult to collect using the source code from the closed-source projects, whereas the usages in bytecode can be collected from their binary files. However, the usages collected from the bytecode are not exactly matched to the hand-written API sequence. The enhance-for statement in Java is an example; this syntax is compiled into a sequence of APIs of interface Iterator and Iterable. Developers do not need to know the compiled API sequence because enhance-for syntax makes shorter code and provides the same functions as the compiled sequence. In this study, API usages were collected from the source code instead of bytecode.

Each method in a Java source code is converted into an API sequence, following the approach of PAM [5]. Figure 5 simplifies the process of collecting API sequences from a method. Each Java source code has at least one class containing multiple methods.

Such a source file is parsed as an abstract syntax tree (AST) using Eclipse JDT parser. Furthermore, this AST is transformed into an API sequence through depth-first traversal started from each method node. As shown in Figure 5, this traversal starts from two methods, C.x() and C.y(), respectively. During the traversal of C.x(), two call sites D.x(a) and D.y(a) are extracted sequentially. Note that the extracted sequences do not consider conditional executions. This approach aims to avoid collecting incomplete sequence such as D.x(a) or D.y(a) only [5]. This example shows how two independent API sequences are extracted from the two methods.

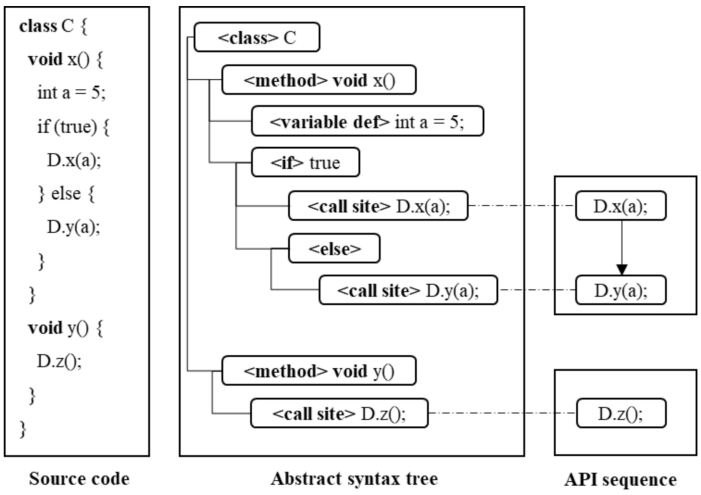

Figure 5. Example of extracting API sequences

5.2. Statistical Model Generator

The statistical model was generalized using the HAPI [17]. HAPI is known to outperform than SLANG [20], which uses n-gram and recurrent neural networks. The HAPI was designed to learn the API usages for each set of classes instead of generating one model for every usage. For example, one model was generated for one class java.util.LinkedList, whereas another model was generated for the class java.util.String. The remainder of this section will explain the HAPI model briefly.

HAPI was employed to approximate the probabilities of state transitions and method invocations. Figure 6 presents the simplified state diagram and a HAPI model of the class MediaPlayer that belongs to the Android standard library. The state diagram presents the methods expected to be invoked from each state. However, such state diagrams are rarely provided with exiting libraries and frameworks including Android. The HAPI aims to generate probabilistic diagrams based on the collected API usages.

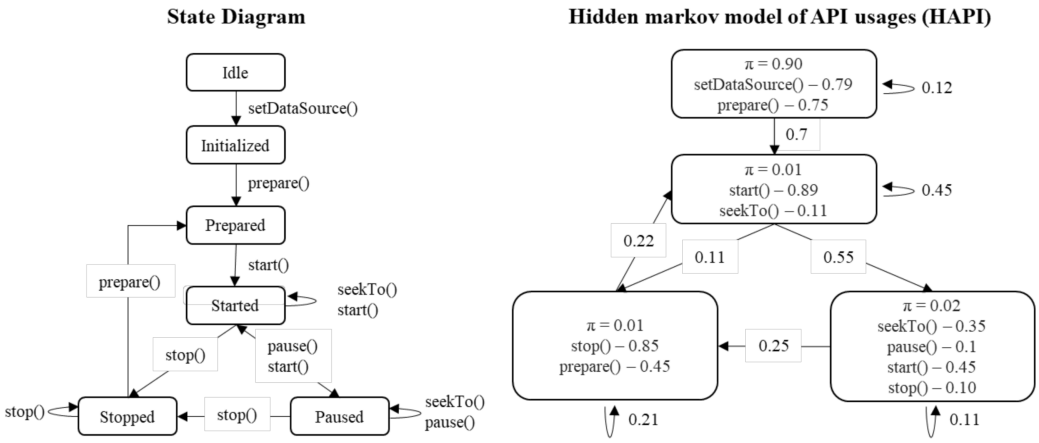

Figure 6. Example of extracting API sequences.

Each HAPI learns the invocation probability of each method in a given set of classes. A HAPI model has k hidden states, $\boldsymbol{H} = \{h_1, \ldots\ h_k\}$; n invocable methods, $\boldsymbol{M} = \{m_1, \ldots\ m_n\}$; a transition matrix $\boldsymbol{A} = \{a_{ij} | i \leq k \wedge j \leq k\}$, where a_{ij} is the transition probability from h_i to h_j; output probability $\boldsymbol{B} = \{b_{ij} | i \leq k \wedge j \leq n\}$, where b_{ij} is the probability of emitting m_j in h_i; and $\boldsymbol{\Pi} = \{\pi_1, \pi_2, \ldots, \pi_k\}$, where π_i is the probability of an API sequence that started from the hidden state h_i. In other words, the HAPI needs to determine the parameters $\lambda = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\Pi})$. This can be solved using the Baum–Welch algorithm [37],

which performs both forward procedures and backward procedures and updates parameters until the parameters converge. Given an API sequence $\boldsymbol{S} = (s_1, \ldots, s_l)$, forward probability $a_{i,t}$ is the probability of sequence $s_1, \ldots, s_t$ when the current state c_t is h_i. Moreover, backward probability $b_{i,t}$ is the probability of sequence $s_{t+1}, \ldots s_l$ when c_t is h_i.

$$a_{i,t} = P(s_1, \ldots, s_t, c_t = h_i|\lambda) b_{i,t} = P(s_{t+1}, \ldots, s_l|c_t = s_i, \lambda)$$

With forward and backward probabilities, we can calculate the transition matrix and emission matrix as follows.

$$\gamma_i(t) = P(c_t = h_i|\boldsymbol{M}, \lambda) = \frac{a_{t,i} b_{i,t}}{P(\boldsymbol{M}|\lambda)} \epsilon_{ij}(t) = P\left(c_t = h_i, c_{t+1} = h_j|\boldsymbol{M}, \lambda\right)$$

Both forward probability and backward probabilities are recalculated until the parameters γ and ϵ converge. Note that among the parameters, the number of hidden states should be determined empirically.

5.3. API Recommendation for Code Completion

The API recommendation based on HAPI computes the likelihood of a method m at location t in a sequence $\boldsymbol{S_t} = \{s_1, s_2, \ldots, s_{t-1}, s_{t+1}, \ldots, s_l\}$. The likelihood represents the possibility that $\boldsymbol{S} = \{s_1, s_2, \ldots, s_{t-1}, m, s_{t+1}, \ldots, s_l\}$ is generated from the HAPI model.

$$P(\boldsymbol{S}, s_t = m|\lambda) = \sum_{i+1}^{K} P(S, s_t = m, c_t = h_i|\lambda) = \sum_{i=1}^{K} a_{i,t} b_{i,t}$$

The recommendation will return the possibility of every method m that can be available at position t. Finally, all the methods are ordered according to the calculated possibility.

6. Experimental Setup

In this section, we explain how we evaluate the API generalization with three research questions.

6.1. Research Questions

RQ1. How many API usages are hidden or fragmented without the generalization?

The generation aims to reveal the hidden usages and fix the fragmented usages. The higher the number of such usages, the more effective the generalization is for improving the coverage and accuracy.

To answer this question, the number of hidden or fragmented usages was counted among the Android- and Java-related API usages collected from the experimental dataset. Assume that an API sequence $\boldsymbol{S}$ is supposed to be collected from a source file. The sequence was considered as fragmented when the generalized usage included any methods that did not exist in the raw usage. Similarly, the sequence was considered as hidden when it could be collected with the generalization only.

RQ2. How accurate is the code completion based on the generalized and raw API usages?

The HAPI models based on the generalized sequences were expected to be more accurate than those based on the raw sequences. The API generalization provided more sequences for the method training because it included the fragmented and hidden sequences. In addition, the generalization helps discover the general API usages that could be found in multiple classes.

To demonstrate this expectation, two different groups of HAPI models were generated based on the generalized and raw API usages, respectively. If the fragmented or hidden APIs were rare, no difference was observed between the accuracies of the two groups. The completion was considered correct when the expected method was ranked in top k.

6.2. Data Collection

The experiments were conducted with 442,809 usages that had Java- or Android-related API calls collected from 2121 projects. All of these projects were a part of the GHTorrent dataset [38]. The API usages were collected from every commit registered between 2014 and 2017. A project was considered as an Android project if its description contained the word term 'android' or 'Android'. In addition, projects that had less than 100 commits were excluded to eliminate unmanaged repositories. The commits that failed to build were also discarded.

The collected API sequences were filtered to obtain the appropriate lengths for the experiments. The minimum and maximum lengths were set to 2 and 15, respectively. The average lengths of the single-class and multiple-class API usages were found to be 3.0 and 7.8, respectively. These averages were similar to the values reported in the HAPI experiments [9].

6.3. Training Code Completion Model

Five-fold cross-validation was adopted to evaluate the accuracy of the code completion. The dataset was divided into five groups; four of which were used as training data and one was selected as test data. The final score was the average of the five iterations.

Furthermore, to determine the number of hidden states for each HAPI model, four-fold cross-validation was conducted with the training data; three of which were used as training data and one was selected as validation data. The validation was repeated with the number of hidden states varying from 1 to 20. For each iteration, the model evaluated the likelihood of the validation data. The number of hidden states was determined to demonstrate the best average likelihood score.

7. Evaluation

7.1. Number of Hidden or Fragmented API Usages

Table 2 introduces four types of API sequences observed in this experiment. As mentioned in Section 6.1, an API sequence was considered hidden when it could be collected only with the proposed generalization approach. In contrast, a sequence was considered fragmented when its generalized form had more than one API call than in the raw sequence. Lastly, the unaffected group consisted of every sequence that was selected regardless of the API generalization.

Table 2. Number of usages in each group.

Sequence Type	Number of Sequences	Number of APIs
Hidden	18,727 (4.2%)	4330 (40.9%)
Unaffected	337,970 (76.3%)	6252 (59.1%)
Fragmented	86,112 (19.4%)	874 (8.2%)

The experimental result demonstrated that the hidden and fragmented usages degraded the coverage of APIs in our dataset by 23.6%. These problematic sequences involved 49.1% of the APIs that appeared in the dataset.

Figure 7 presents a fragmented sequence in our experiment. The source code invokes three methods, setVisibility(), setEnabled(), and setText() with five variables.

As shown in Figure 7, the raw sequence extracted without the generalization ignored three methods that were invoked with a non-Android library class, WPTextView. In contrast, the API generalization detected the methods invoked with mProgressTextSignIn and mSignupButton, and could further convert them to TextView. In this case, the raw sequence was considered fragmented.

Figure 8 illustrates a hidden sequence in our experiment. This code aimed to instantiate an option menu within a developer-written class. It can be seen that the raw sequence ignored two methods invoked with the non-library class WPWebViewActivity. As a result, this sequence was discarded as its

length was less than two, which was the minimum length in our study. In contrast, the generalization detected that WPWebViewActivity could be converted to the Android class Activity. In this case, the raw sequence was considered hidden.

Source Code	Raw Sequence (w/o generalization)	Generalized Sequence
public void startProgress(String message) {		
mProgressBarSignIn.setVisibility(View.VISIBLE);	RelativeLayout.setVisibility	View.setVisibility
mProgressTextSignIn.setVisibility(View.VISIBLE);	~~WPTextView.setVisibility~~	TextView.setVisibility
mSignupButton.setVisibility(View.GONE);	~~WPTextView.setVisibility~~	TextView.setVisibility
mProgressBarSignIn.setEnabled(false);	RelativeLayout.setEnabled	View.setEnabled
mProgressTextSignIn.setText(message);	~~WPTextView.setText~~	TextView.setText
mSiteTitleTextField.setEnabled(false);	EditText.setEnabled	View.setEnabled
mSiteUrlTextField.setEnabled(false);	EditText.setEnabled	View.setEnabled
}		

Figure 7. Example of a fragmented API sequence (strikethrough on non-Android APIs).

Source Code	Raw Sequence (w/o generalization)	Generalized Sequence
public boolean onCreateOptionsMenu(Menu menu) {		
super.onCreateOptionsMenu(menu);	~~WPWebViewActivity.onCreateOptionsMenu~~	Activity.onCreateOptionsMenu
MenuInflater inflater = getMenuInflater()	~~WPWebViewActivity.getMenuInflater~~	Activity.getMenuInflater
inflater.inflate(R.menu.webview, menu);	MenuInflator.inflate	MenuInflator.inflate
return true;		
}		

Figure 8. Example of a hidden API sequence (strikethrough on non-Android APIs).

7.2. Accurate of Code Completion Based on the Generalized and Raw API Usages

Figure 9 presents the accuracies of the HAPI based on raw and generalized API sequences. An HAPI that learned the API usages of one class only was categorized as a single-class model. It was categorized as a multi-class model, otherwise. These two categorizations are presented separately because the multi-class models would be relatively more difficult to learn than the single-class ones. This separation helped to investigate the impact of the generalization, independent of the prediction difficulty. In addition, code completion was conducted only for the "unaffected" and "fragmented" sequences as the models based on the raw sequences were incapable of observing the hidden usages.

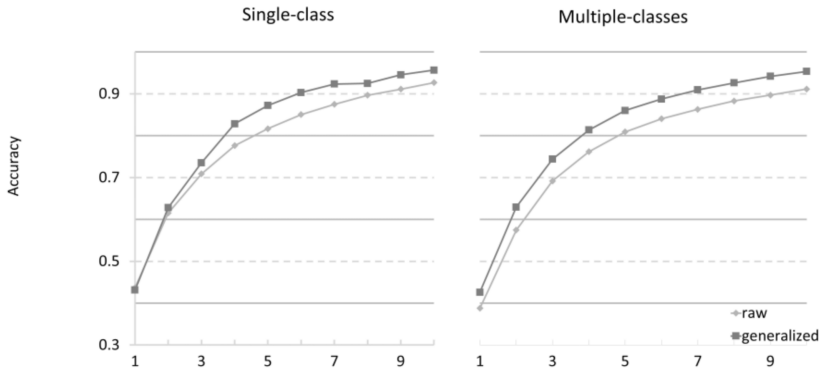

Figure 9. Comparison between raw and generalized APIs.

The accuracies approximately equaled at the top-one case, which was 0.52% in the single-class models. The increment at the high-top case was expected as the prediction became easier to be accurate with a high number of API candidates. Nonetheless, the models based on the generalized APIs tended to be more accurate than those based on the raw ones, up to 5.55%. This tendency was similar even in

the multi-class usages, which was the highest (5.47%) at the top-two case. In summary, the generation was effective in improving the code completion accuracy.

8. Discussion and Threats to Validity

The fragmented or hidden API usages accounted of 23% in our dataset. Such problematic usages tended to contain the APIs inherited from Android library classes. Table 3 presents the top three APIs that appeared in the hidden and fragmented usages. For example, the most popular API in each problematic usage was setContentView() and supportStartPostponedEnterTransition(), respectively. These APIs were defined in the Android class Activity, which aims to manage the lifecycle and user interfaces in Android applications. The other methods in Table 3 were also defined in the Android classes that were inherited by the developer-written classes. These results supported that the inherited methods incurred hidden or fragmented usages.

The influence of fragmented and hidden usages on code completion was significant. The accuracy was improved by at most 5.55% with the API generalization. A feasible explanation is that the API generalization reduces the number of observable outputs in each HAPI model. For example, in Figure 6, the raw sequence consisted of five methods, setVisibility() and setEnabled() of RelativeLayout, setEnabled() of EditText, and setVisibility() and setText() of WPTextView. In contrast, the generalized sequence consisted of four methods, setVisibility() and setEnabled() of View and TextView, respectively. This difference helped the HAPI model consider fewer methods and classes.

Table 3. Most frequently observed APIs in each group.

Group	API Ranking
Hidden	1. AppCompatActivity.setContentView() 2. CursorWrapper.getColumnIndex() 3. Fragment.setHasOptionMenu()
Fragmented	1. FragmentActivity.supportStartPostponedEnterTransition() 2. CursorLoader.setSelection() 3. CursorLoader.setSelectionArgs()

Another feasible explanation is that the fragmented usages degraded the accuracy of the code completion. Figure 10 presents the accuracy of the code completion only for the fragmented usages, all of which consisted of multiple classes.

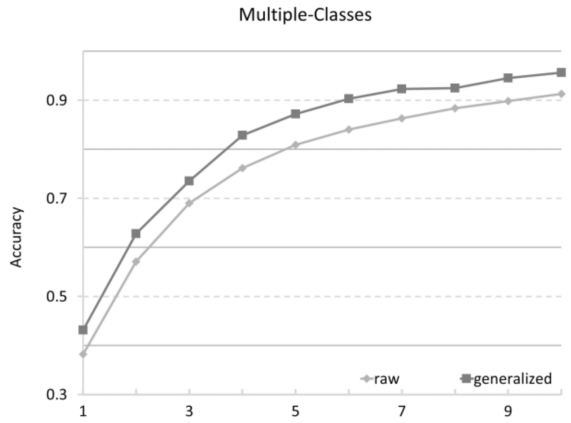

Figure 10. Comparison between raw and generalized APIs.

Regardless of the threshold, the accuracy was always higher with the generalized usages than with the raw usages. The maximum difference was 6.66% observed at the top two recommendation,

whereas the minimum was 4.08% observed at the top eight. This result implies that the API usages learned from the generalized usages were more predictable than the fragmented usages.

Our study focused on the code completion task. However, API usage mining has been used in various applications such as code search or bug prediction. The proposed API generalization would influence such applications. In addition, it is necessary to confirm whether the generalization was effective even with the larger dataset and on various platforms other than Android and Java.

A threat to validity could be introduced due to the possibility of bugs in the mining tools, such as Eclipse JDT, that were used in our experiments. Similarly, the class hierarchy may have been resolved incorrectly due to the bugs in the build tool or Eclipse JDT. We used the most recent build version in which several bugs have been corrected.

9. Conclusions

The API usages were extracted from the source code without considering that they could be utilized in similar classes. In this paper, we proposed an approach that aims to generalize the API usages and investigate whether the usages could be hidden or fragmented without the generalization.

The three experiments demonstrated that 23% of API sequences were hidden or fragmented in Android projects. In addition, the accuracy of the code completion was improved by up to 6.66% when the usages were generalized. In our future works, we aim to conduct experiments on various dataset, platforms, and languages to confirm the effectiveness of the generalization.

Author Contributions: Software and writing—original draft preparation, M.K.; writing—review and editing, N.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2020R1C1C1014611).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Robillard, M.P. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Softw.* **2009**, *26*, 27–34. [CrossRef]
2. Michail, A. Data mining library reuse patterns using generalized association rules. In Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, 9 June 2000; pp. 167–176.
3. Nguyen, A.T.; Nguyen, T.N. Graph-Based Statistical Language Model for Code. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; pp. 858–868.
4. Xie, T.; Pei, J. MAPO: Mining API usages from open source repositories. In Proceedings of the International Workshop on Mining Software Repositories, Shanghai, China, 20–28 May 2006; pp. 54–57.
5. Fowkes, J.; Sutton, C. Parameter-free probabilistic API mining across GitHub. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 18 November 2016; pp. 254–265.
6. Buse, R.P.L.; Weimer, W. Synthesizing API usage examples. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 782–792.
7. Wu, D.; Jing, X.; Zhang, H.; Kong, X.; Xie, Y.; Huang, Z. Data-driven approach to application programming interface documentation mining: A review. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **2020**, *10*, 1–28. [CrossRef]
8. Zhong, H.; Xie, T.; Zhang, L.; Pei, J.; Mei, H. MAPO: Mining and Recommending API Usage Patterns. In Proceedings of the European Conference on Object-Oriented Programming, Genoa, Italy, 3–10 July 2009; pp. 318–343.
9. Wang, J.; Dang, Y.; Zhang, H.; Chen, K.; Xie, T.; Zhang, D. Mining succinct and high-coverage API usage patterns from source code. In Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, 18–19 May 2013; pp. 319–328.

10. Nguyen, T.T.; Nguyen, H.A.; Pham, N.H.; Al-Kofahi, J.M.; Nguyen, T.N. Graph-based mining of multiple object usage patterns. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Amsterdam, The Netherlands, 24–29 August 2009; pp. 383–392.
11. Moreno, L.; Bavota, G.; Di Penta, M.; Oliveto, R.; Marcus, A. How Can I Use This Method? In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; pp. 880–890.
12. Nguyen, P.T.; Di Rocco, J.; Di Ruscio, D.; Ochoa, L.; Degueule, T.; Di Penta, M. FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 1050–1060.
13. Yuan, W.; Nguyen, H.H.; Jiang, L.; Chen, Y.; Zhao, J.; Yu, H. API recommendation for event-driven Android application development. *Inf. Softw. Technol.* **2019**, *107*, 30–47. [CrossRef]
14. Proksch, S.; Lerch, J.; Mezini, M. Intelligent Code Completion with Bayesian Networks. *ACM Trans. Softw. Eng. Methodol.* **2015**, *25*, 1–31. [CrossRef]
15. Robbes, R.; Lanza, M. How Program History Can Improve Code Completion. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, 15–19 September 2008; pp. 317–326.
16. Bruch, M.; Mezini, M.; Monperrus, M. Mining subclassing directives to improve framework reuse. In Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010; pp. 141–150.
17. Nguyen, T.T.; Pham, H.V.; Vu, P.M.; Nguyen, T.T. Learning API usages from bytecode: A statistical approach. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 416–427.
18. Saied, M.A.; Abdeen, H.; Benomar, O.; Sahraoui, H. Could We Infer Unordered API Usage Patterns Only Using the Library Source Code? In Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, Florence, Italy, 18–19 May 2015; pp. 71–81.
19. Ogasawara, K.; Kanda, T.; Inoue, K. On the variations and evolutions of API usage patterns: Case study on Android applications. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, Seoul, Korea, 23–29 May 2020; pp. 746–753.
20. Raychev, V.; Vechev, M.; Yahav, E. Code completion with statistical language models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, UK, 9–11 June 2014; pp. 419–428.
21. Nguyen, T.T.; Nguyen, A.T.; Nguyen, H.A.; Nguyen, T.N. A statistical semantic language model for source code. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, 18–26 August 2013; pp. 532–542.
22. Allamanis, M.; Barr, E.T.; Bird, C.; Sutton, C. Learning natural coding conventions. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–22 November 2014; pp. 281–293.
23. Yan, J.; Qi, Y.; Rao, Q.; He, H.; Qi, S. LSTM-Based with Deterministic Negative Sampling for API Suggestion. *Int. J. Softw. Eng. Knowl. Eng.* **2019**, *29*, 1029–1051. [CrossRef]
24. Maddison, C.J.; Tarlow, D. Structured generative models of natural source code. In Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 21–26 June 2014; pp. 649–657.
25. Gu, X.; Zhang, H.; Kim, S. CodeKernel: A graph kernel based approach to the section of API usage examples. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, San Diego, CA, USA, 11–15 November 2019; pp. 590–601.
26. Nguyen, T.T.; Vu, P.M.; Nguyen, T.T. Code Search on Bytecode for Mobile App Development. In Proceedings of the 2019 ACM Southeast Conference, Kennesaw, GA, USA, 18–20 April 2019; pp. 253–256.
27. Nguyen, T.T.; Pham, H.V.; Vu, P.M.; Nguyen, T.T. Recommending API Usages for Mobile Apps with Hidden Markov Model. In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 795–800.
28. Zhong, H.; Zhang, L.; Mei, H. Inferring Specifications of Object Oriented APIs from API Source Code. In Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference, Beijing, China, 2–5 December 2008; pp. 221–228.

29. Schäfer, T.; Jonas, J.; Mezini, M. Mining framework usage changes from instantiation code. In Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; pp. 471–480.
30. Gu, X.; Zhang, H.; Zhang, D.; Kim, S. Deep API learning. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 631–642.
31. Moritz, E.; Linares-Vásquez, M.; Poshyvanyk, D.; Grechanik, M.; McMillan, C.; Gethers, M. ExPort: Detecting and visualizing API usages in large source code repositories. In Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, USA, 11–15 November 2013; pp. 646–651.
32. Cottrell, R.; Walker, R.J.; Denzinger, J. Semi-automating small-scale source code reuse via structural correspondence. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, GA, USA, 9–14 November 2008; pp. 214–225.
33. Nguyen, A.T.; Hilton, M.; Codoban, M.; Nguyen, H.A.; Mast, L.; Rademacher, E.; Nguyen, T.N.; Dig, D. API code recommendation using statistical learning from fine-grained changes. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 511–522.
34. Nguyen, A.T.; Nguyen, T.T.; Nguyen, H.A.; Tamrawi, A.; Nguyen, H.V.; Al-Kofahi, J.; Nguyen, T.N. Graph-based pattern-oriented, context-sensitive source code completion. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 69–79.
35. Gosling, J.; Joy, B.; Steele, G.; Bracha, G.; Buckley, A.; Smith, D. *Java Language Specification*, Java SE 11 ed.; Oracle America, Inc.: Redwood City, CA, USA, 2018; pp. 257–270.
36. Lindholm, T.; Yellin, F.; Bracha, G.; Buckley, A.; Smith, D. *The Java Virtual Machine Specification*, Java SE 12 ed.; Oracle America, Inc.: Redwood City, CA, USA, 2019; pp. 368–389.
37. Rabiner, L.; Juang, B. An introduction to hidden Markov models. *IEEE ASSP Mag.* **1986**, *3*, 4–16. [CrossRef]
38. Gousios, G. The GHTorrent dataset and tool suite. In Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013; pp. 233–236.