2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Learning API Usages from Bytecode: A Statistical Approach

Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, Tung Thanh Nguyen
Computer Science Department
Utah State University
{tam.nguyen,hung.pham,phong.vu}@aggiemail.usu.edu
tung.nguyen@usu.edu

ABSTRACT

Mobile app developers rely heavily on standard API frameworks and libraries. However, learning API usages is often challenging due to the fast-changing nature of API frameworks for mobile systems and the insufficiency of API documentation and source code examples. In this paper, we propose a novel approach to learn API usages from bytecode of Android mobile apps. Our core contributions include HAPI, a statistical model of API usages and three algorithms to extract method call sequences from apps' bytecode, to train HAPI based on those sequences, and to recommend method calls in code completion using the trained HAPIs. Our empirical evaluation shows that our prototype tool can effectively learn API usages from 200 thousand apps containing 350 million method sequences. It recommends next method calls with top-3 accuracy of 90% and outperforms baseline approaches on average 10-20%.

Keywords

Statistical model, API usage, mobile apps

1. INTRODUCTION

Mobile apps are software applications developed specially for mobile devices like smartphones and tablets. Due to the exploding in popularity and usage of those devices, millions of mobile apps have been developed and made available to end-users. Due to the fierce competition, those mobile apps often need to have very short time-to-market and upgrade cycles, and thus, short development time. To address this requirement, mobile app developers often rely heavily on API application frameworks and libraries such as Android and iOS frameworks, Java APIs, etc and mobile apps extensively re-use API components. For example, a prior study reports some Android apps having up to 42% of their external dependencies to Android APIs and 68% to Java APIs [38].

Learning API usages is usually challenging due to several reasons. First, an API framework often consists a large number of components. For example, the Android application framework contains over 3,400 classes and 35,000 methods which are organized in more than 250 packages [23]. Moreover, while common API usage scenarios often involve several API elements and follow special rules for pre- and post-conditions or for control and data flows [31, 28, 10], documentation of such usages is generally insufficient. For example, the Javadoc of a class mainly contains only descriptions of its fields and methods and rarely has code examples describing in details the usages of its objects and methods [22]. Descriptions and code examples for API usages involving several objects are usually non-existed.

These challenges are even more severe for learning APIs of mobile frameworks. Due to the fast development of mobile devices and the strong competition between software and hardware vendors, those frameworks are often upgraded quickly and include very large changes. For example, 17 major versions of Android framework containing nearly 100,000 method-level changes have been released within five years [23]. Additionally, source code of most mobile apps is not publicly available. With few apps with source code available, finding and learning code examples from existing mobile app projects would be difficult and insufficient.

In this paper, we introduce SALAD (*"Statistical Approach for Learning APIs from DVM bytecode"*), a novel approach to address the problem of learning API mobile frameworks. To address the problem of insufficient documentation and source code examples, SALAD learns API usages from *bytecode* of Android mobile apps, of which millions are publicly available. As a statistical approach, SALAD can learn complex API usages involving several API objects and methods. Finally, SALAD can automatically generate recommendations for incomplete API usages, thus it could reduce the chance of API usage errors and improve code quality.

The key component of our approach is HAPI, standing for *"Hidden Markov Model of API usages"*. A HAPI is a Hidden Markov Model (HMM) [34] representing method call sequences involving one or multiple related API objects. It has several states aimed to represent the internal states of the represented API objects. It also associates with probabilities for selecting the starting state, for transiting from one state to another, and for calling a method at a given state. Those probabilities describe the statistical patterns of API states and method calls.

SALAD also consists of three new algorithms involving HAPI. One algorithm is designated to train a HAPI i.e. inferring its internal states and estimating the associated probabilities from a very large collection of method call se-

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884873

quences. Another algorithm uses a trained HAPI to compute the generating probabilities of several method sequences and rank those sequences based on those probabilities. This ranking result is used for API usage recommendation.

The last algorithm extracts API method sequences from apps' bytecode which are then used for training HAPIs. It first analyzes bytecode and builds GROUMs. A GROUM (*"GRaph-based Object Uasge Model"*) [31] is a graph-based model in which the nodes represent method calls and data objects and the edges indicate control and data flows between those methods and objects. With that design, a GROUM can represent an usage scenario including many objects and methods and complex control and data flows between them. Once a GROUM is built, our algorithm can travel all its paths following the control and data dependencies and extracts the corresponding method sequences.

We have conducted several experiments to evaluate the usefulness and effectiveness of SALAD. The experiment results show several important points. First, our approach SALAD is highly efficient and scalable. It is able to extract nearly 350 millions method sequences and train 24 thousands HAPIs from more than 200 thousands mobile apps downloaded from Google's Android app market. With that huge amount of training data, SALAD can predict and recommend the next method call for a given API method call sequence with a top-3 accuracy of nearly 90% and a top-10 accuracy of more than 98%. In addition, our model HAPI consistently outperforms two baseline models *n*-gram and recurrent neural network [35] with the average improvement around 10-20%.

Based on SALAD, we have developed an API usage recommendation tool named DroidAssist. This tool is implemented as a plug-in of Android Studio and available online at http://useal.cs.usu.edu/droidassist. More information about this tool can be found in [32] and that website. Source code and experiment data of this work (e.g. extracted API method sequences, GROUMs, and HAPIs) are also made available there.

The remaining of this paper is organized as the following. Section 2 presents an example of Android APIs and their usages in practice. It also illustrates how HAPIs could be used to model API usages. Section 3 introduces HAPI as the main conceptual contributions of SALAD. Section 4 discusses the overall architecture of SALAD and its key algorithms. Our empirical evaluation is presented in Section 5. We discuss related work in Section 6.

2. EXAMPLE

This section briefly introduces API usages and HAPI via an example. Figure 1, reproduced from Android Developer website [18], illustrates usages of a MediaRecorder object in Android API framework as a state diagram. This state diagram is a finite state machine in which each node (drawn as a rounded rectangle) represents an internal state of the MediaRecorder object and each edge (drawn as an arrow) represents a state transition when a method (drawn as the label of the edge) is called.

We learn from this state diagram that a MediaRecorder object has seven states during its lifetime. It is at one state at a time and can change to another state if a method is called. For example, as after being created, the MediaRecorder object is in the *Initial* state. If method setAudioSource or setVideoSource is called, it changes to the *Initialized* state.

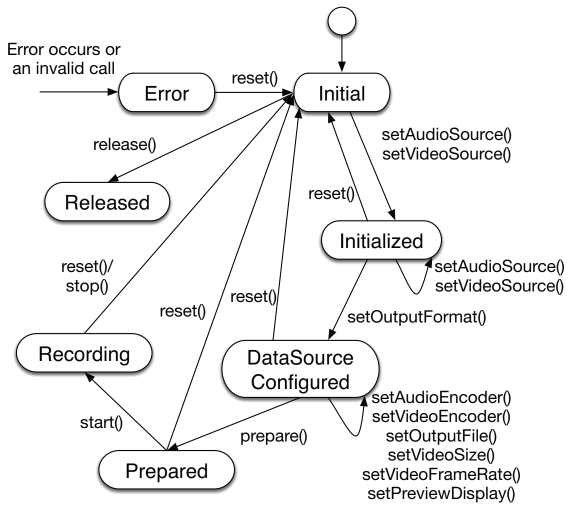

Figure 1: State diagram of MediaRecorder object

Then, it will change from the *Initialized* state to the *DataSourceConfigured* state if we call method setOutputFormat. However, at any time, if method reset is called, the object will come back to its *Initial* state.

It could be seen from this example that state diagrams can represent usages of API objects precisely and concisely. Thus, they are useful to learn and understand API usages. For example, we could infer from Figure 1 the following method call sequence to perform an audio recording task.

```
setAudioSource(...)
setOutputFormat(...)
setAudioEncoder(...)
setOutputFile(...)
prepare()
start()
stop()
release()
```

Unfortunately, documentation of most of API objects does not contain state diagrams representing their usages. In addition, state diagrams can be used to to check for the *validity* of method call sequences, but not their *popularity*. For example, we cannot know from a state diagram that a sequence is *valid* but *has never been used*, or compare if a sequence *is more likely to be used* in practice than another. Last but not least, current code completion engines like the the built-in engine in Android Studio IDE do not incorporate state diagrams in their recommendations.

We design HAPI to take advantages of the usefulness of state diagrams in representing API usages and address their discussed limitations. In essential, a HAPI is a *probabilistic state diagram*. Its nodes and edges associate with probabilities which can be used to estimate the probability of any method call sequences. More importantly, its structure and probabilistic parameters can be learned directly from data.

Figure 2 illustrates a HAPI model learned by SALAD to represent the usages of a MediaRecorder object. In this figure, each node of the HAPI represents a state of the object. But different from the state diagram in Figure 1, each state of a HAPI has a probability π for being the starting state in a method sequence. It also has a distribution specifying the probability to call a particular method (i.e. emission probabilities) and another distribution specifying the prob-

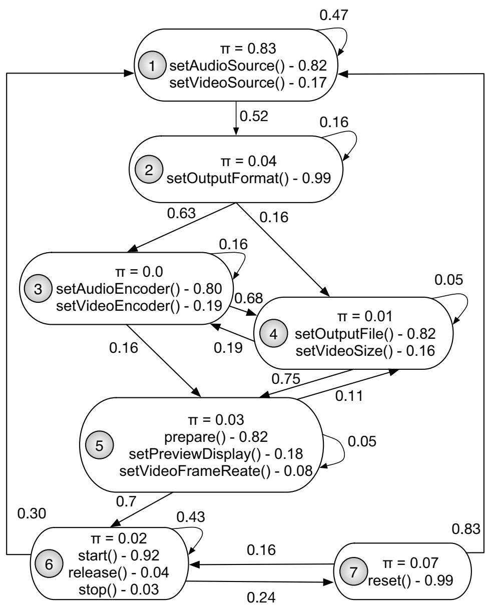

Figure 2: HAPI model for usages of MediaRecorder object

ability to change to another state (i.e. transition probabilities). To simplify the drawing, we only show method calls and transition edges with significant probabilities in the figure. By design, the states of a HAPI model aim to represent the internal states of the corresponding API objects. The transition probabilities capture switching patterns of those internal states; and the emission probabilities describe the method calling patterns at each of those states.

For example, accordingly to the HAPI model in Figure 2, a MediaRecorder object starts in state (1) with probability of 83%. In this state, method setAudioSource can be called with a probability of 82% and when that happens, the object can change to state (2) with a probability of 52%. In state (2), setOutputFormat can be called with a probability of 99% and so on. Technically, a HAPI is a Hidden Markov Model [34] which can be trained, i.e. inferring its structure and the associating probabilities, from a given collection of method call sequences. Once trained, we can use it to compute and compare the probabilities of any given method sequences. For example, the sequence setAudioSource, setOutputFormat, setAudioEncoder would have higher probability than the sequence setAudioSource, setOutputFormat, start.

HAPI models have several advantages over plain state diagrams. First, the state diagram only specifies the general rules for using objects but not specify the common usages. In contrast, once learned from code examples, HAPI could infer common usages of an API object by searching for the method sequences that have highest probabilities. Second, a HAPI could be used to predict the most likely next method call in a given API method call sequence. Thus, we can use HAPI to recommend API usages while developers are writing code.

3. API USAGE MODEL

In this section, we will discuss HAPI, our proposed statistical model for API usages in more details. We also re-

```
protected void startRecording(String file) {
 MediaRecorder recorder = new MediaRecorder();
 recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
 recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
 recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
 recorder.setOutputFile(file);
 try {
   recorder.prepare();
 } catch (IllegalStateException e) {
   e.printStackTrace();
 } catch (IOException e) {
   e.printStackTrace();
 }
 recorder.start();
}
```

Figure 3: Code example using MediaRecorder object

introduce GROUM, a graph-based model for object usages developed previously by Nguyen *et al.* [31]. SALAD uses GROUM as a temporary representation for API usages extracted from mobile apps' bytecode.

3.1 HMM-based API Usage Model

A HAPI is a generative, probabilistic model that describes the process of generating method call sequences involving one or multiple API objects. As a Hidden Markov Model [34], a HAPI formally has a set $Q = \{q_1, q_2, ..., q_K\}$ of K hidden states and associates with a set $V = \{v_1, v_2, ..., v_M\}$ of M API methods of the API object(s) it is modeling. Each state q_i has a probability π_i to be selected as the starting state of the model. When being in one state, a HAPI can emit (i.e. generate) a method call and/or switch to another state. The transition matrix $A = \{a_{ij} | i, j \in 1..K\}$ specifies the state transition probabilities. That means, a_{ij} is the probability that the model changes from state q_i to state q_j. The generating matrix $B = \{b_{ik} | i \in 1..K, k \in 1..M\}$ specifies the emission probabilities. Specially, b_{ik} is the probability to call method v_k when the model is in state q_i. As seen, this HAPI model has $K + K^2 + KM$ parameters.

With those parameters, the HAPI model can generate a method sequence $Y = (y_1, y_2, .., y_T)$ via the following generating process:

1. Set $t = 1$ and sample the state q_{i_t} from the initial state distribution $\pi = \{\pi_1, \pi_2, ..., \pi_K\}$

2. Sample a method call y_t from the calling distribution of state q_{i_t}, i.e. $\{b_{i_t 1}, b_{i_t 2}, ..., b_{i_t M}\}$

3. Sample the next state $q_{i_{t+1}}$ from the transition distribution of state q_{i_t}, i.e. $\{a_{i_t 1}, a_{i_t 2}, ..., a_{i_t K}\}$

4. Increase t and loop back step 2 for T iterations.

Modeling API usages using HAPI involves two problems. The first problem is training (i.e. estimate the parameters of) a HAPI model for one or multiple API objects from existing method call sequences involving. The second problem is using the trained HAPI models to compute the generating probabilities of any method call sequences for recommending API usages. In Section 4, we will present two algorithms we developed in SALAD to solve those two problems.

3.2 Graph-based Object Usage Model

A HAPI model for one or multiple API objects is trained by a collection of method sequences involving those objects. To extract those sequences from bytecode, we employ GROUM, a graph-based model of object usages [31], as a temporary representation of API usages.

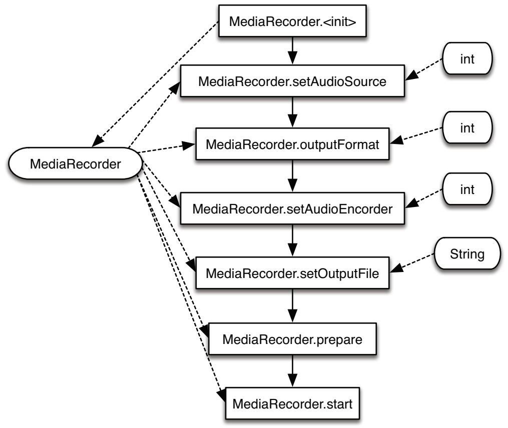
MediaRecorder.<init>
MediaRecorder.setAudioSource
int
MediaRecorder.outputFormat
int
MediaRecorder
MediaRecorder.setAudioEncorder
int
MediaRecorder.setOutputFile
String
MediaRecorder.prepare
MediaRecorder.start

Figure 4: GROUM model for usages of MediaRecorder object

Figure 3 lists a code example with the usage of a MediaRecorder object. Figure 4 illustrates the GROUM built for the main execution path of that code example. As seen in this figure, a GROUM is a labeled, directed graph. It has two kinds of nodes: *object nodes* and *action nodes*. An object node represents an object. It is labeled by the name of the object type (e.g. MediaRecorder) and illustrated as a rounded rectangle in the figure. An action node represents a method call. It is labeled the method qualified name (e.g. MediaRecorder.start) and illustrated as a flat rectangle. There are two kinds of edges representing *control flow* and *data flow*. In the figure, solid arrows illustrate control flow edges between action nodes while dashed arrows illustrate data flow edges between object nodes and other nodes.

Using GROUM to represent API usage scenarios has two main advantages. First, we can easily identify all action nodes having data-dependency with a given object node or set of object nodes, from which we can extract method sequences. Second, we could determine if a given set of object nodes have usage-dependency. A set of object nodes has usage-dependency if there is at least one action node that have data-dependency with all of them. SALAD only extracts method sequences of usage-dependent objects because they are more likely to represent legitimate API usages involving multiple objects.

4. SYSTEM ARCHITECTURE

Figure 5 shows the overall architecture and processing pipeline of SALAD. For the pre-processing phase, it has two components to extract GROUMs from bytecode and source code and another component to extract API method sequences from those GROUMs. In the training phase, the HAPI Learner component is responsible to use those method sequences to train the HAPI models. Once trained, those models can use two components Method Call Recommender and Method Sequence Analyzer to recommend a next method calls for a given method sequence and to estimate the likelihood of that sequence, respectively. In the rest of this section, we will present the design and implementation details of those components.

4.1 Bytecode GROUM Extractor

This component is responsible to extract GROUMs from

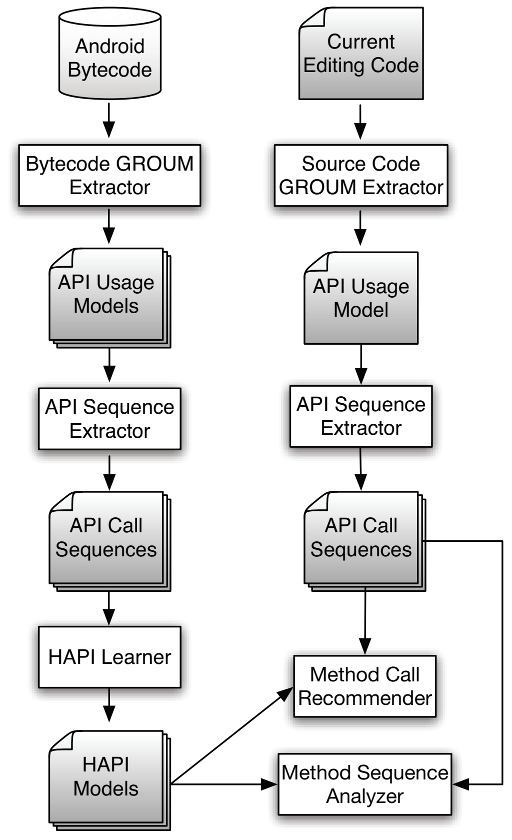

Figure 5: The overview of SALAD's architecture

apps' bytecode instructions. Bytecode of Android apps is stored as .dex files and available on online app markets like Google Play Store. Thus, the extraction process has three steps: 1) Downloading the .dex files, 2) Parsing those files into control flow graphs (CFGs), and 3) Analyzing those CFGs to build the GROUMs.

4.1.1 Downloading .dex files

SALAD has a built-in app crawler developed based on the Google Play Crawler project [20]. This crawler simulates an Android device, thus it can access the app store like a normal smartphone. We program it to download all the top free/new free apps in all categories. For each download application package (.apk files), we extract and store the .dex files which contain all of its bytecode.

4.1.2 Parsing .dex files

The next step is to parse .dex files to build control flow graphs. SALAD employs smali [19], an assembler/disassembler for Android .dex files, to obtain bytecode instructions implemented for each class and method available in the .dex files. Based on those instructions, SALAD constructs a Control Flow Graph (CFG) for each method to simplify the further analysis tasks. Let us discuss this in more details.

Dalvik Virtual Machine. Android apps are originally developed in Java and re-targeted for execution in Dalvik Virtual Machine (DVM). DVM is a register-based virtual machine. It has a set of 32-bit registers for storing primitive values (e.g. integers and floating point numbers) and object

```
public String readTextFile(String fileName) throws IOException {
        FileReader fr = new FileReader(fileName);
        BufferedReader br = new BufferedReader(fr);
        StringBuilder strBuilder = new StringBuilder();
        String line;
        while((line = br.readLine()) != null) {
                strBuilder.append(line);
        }
        br.close();
        return strBuilder.toString();
}
```

Figure 6: A source code example

```
|[0001dc] IOManager.readTextFile:(Ljava/lang/String;)Ljava/lang/String;
|0000: new—instance v1, Ljava/io/FileReader;
|0002: invoke—direct {v1, v6}, Ljava/io/FileReader;.<init>:(Ljava/lang/String;)V
|0005: new—instance v0, Ljava/io/BufferedReader;
|0007: invoke—direct {v0, v1}, Ljava/io/BufferedReader;.<init>:(Ljava/io/Reader;)V
|000a: new—instance v3, Ljava/lang/StringBuilder;
|000c: invoke—direct {v3}, Ljava/lang/StringBuilder;.<init>:()V
|000f: invoke—virtual {v0}, Ljava/io/BufferedReader;.readLine:()Ljava/lang/String;
|0012: move—result—object v2
|0013: if—nez v2, 001d
|0015: invoke—virtual {v0}, Ljava/io/BufferedReader;.close:()V
|0018: invoke—virtual {v3}, Ljava/lang/StringBuilder;.toString:()Ljava/lang/String;
|001b: move—result—object v4
|001c: return—object v4
|001d: invoke—virtual {v3, v2}, Ljava/lang/StringBuilder;.append:(Ljava/lang/String;)Ljava/lang/
          StringBuilder;
|0020: goto 000f
```

Figure 7: Dalvik bytecode of code example in Figure 6

references. A frame (activation record) of a method has a fixed size and consists of a particular number of registers for storing local variables, parameters (including the this reference), return values, and temporary values.

Figure 6 lists a code snippet and Figure 7 shows the Dalvik bytecode compiled for the method readTextFile in that snippet. For execution, this method is allocated 7 registers v0-v6. v5 is for the receiver object (i.e. this). v6 is for the parameter, a String object for the file name. Registers v0-v4 are used for local or temporary variables. Instructions in Dalvik bytecode operate on registers. For example, instruction mul-int v2,v5,v3 multiples the values of registers v2 and v5 and stores the result to v3. Or instruction new-instance v1, Ljava/io/FileReader; creates a new FileReader object and returns the reference of that object to register v1.

Control Flow Graph. We construct Control Flow Graph (CFG) as a representation of bytecode instructions. A node in the constructed CFG contains a single bytecode instruction and an edge is the control flow between the two instructions. There are two types of nodes in CFG. Control nodes represent control instructions in bytecode, e.g. if, return, throw, or goto instructions. Other instructions are normal nodes. Normal nodes in a CFG only have one out-going edge points to the next instruction while control nodes could have several out-going edges (if, switch nodes) or do not have any (return nodes). Techniques used for constructing CFGs are quite standard. First, we create all nodes in the CFG, each one corresponds to an instruction in the instruction list. Then, for each node, we use offsets to identify nodes that are executed after it and add edges from the current node to those nodes.

4.1.3 Building GROUM

In this step, SALAD takes a CFG as input and produce GROUMs which describes usage scenario for each execution

```
1   function BuildGROUM(Method $M$)
2     $CFG$ = BuildCFG($M$)
3     $A = \emptyset$ //list of GROUMs
4     $S_t$ = CreateStartState($CFG, M$)
5     $F = \emptyset$
6     Push($F, S_t$)
7     while $F \neq \emptyset$
8       $S$ = Pop($F$)
9       $SN$ = GetStartNode($S$)
10      while $SN$ is not a control node
11        BuildTemporaryGROUM($S, SN$)
12        AddExploredNode($S, SN$)
13        $CN$ = GetNextNode($SN$)
14      if $SN$ is the return node
15        AddGROUM($A, S$)
16      else
17        for $NN \in$ GetNextNodes($SN$)
18          if $NN \notin$ GetExploredNodes($S$)
19            $S_c$ = GetCopy($S$)
20            SetStartNode($S_c, NN$)
21            Push($F, S_c$)
22    return $A$
```

Figure 8: Building GROUM Algorithm

path. The main idea of the algorithm to construct GROUM is to explore all the execution path in CFG and build temporary GROUMs when exploring those paths. Once a path has been explored, it collects the GROUM that have been built for that path. Each CFG has a start node which is the first instruction and a termination node which is the return node. Our algorithm needs to find all execution paths from the start node to the termination node. One problem that the algorithm needs to consider is to handle loops that occur in the CFG. These loops represent while or for loops in source code. The instructions inside loops may be executed either several times or zero time, thus, could lead to infinity number of execution paths. On the other hand, considering these instructions once can help build usage scenario associated with loops. Therefore, our algorithm consider instructions inside a loop to be executed either once or none. In other words, a loop is executed at most once.

Detailed Algorithm. Figure 8 show the pseudo-code for our algorithm. Input of the algorithm is the bytecode of a method M. The algorithm starts with creating CFG from the bytecode instructions using techniques described the previous section (line 1). Similar to depth first search algorithm, we maintain a stack F to store states which are frontiers to explore (line 4). Each state of our algorithm represents an incomplete execution path and contains following information: 1) start node: the current node in CFG when a state is pop from stack F, 2) explored nodes: a set of all nodes in CFG that have been visited 3) a temporary GROUM. The initial state S_t is created in line 3 with start node is the begin node of CFG, explored nodes and the temporary GROUM of this state are empty. The stack F is initialized with S_t in line 4. In the main while loop of the algorithm, each time, a state S is pop from F. We start with SN, which is the start node of S (line 8) and explore the path from SN to the next control node in CFG. Whenever SN is a normal node in CFG, we use SN to build and update the temporary GROUM of S (we will describe the algorithm to build a temporary GROUM in the next section). We then add SN to the set of explored nodes of S and update SN equal to the next node of SN in CFG (because SN is still a normal node in CFG, it only has one

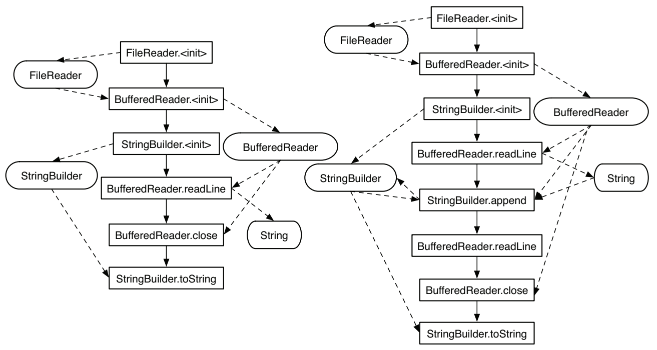

Figure 9: GROUMs

next node). After the loop from line 9 to line 12. SN now is a control node in CFG. If SN is the return node of CFG then we have explored one execution path of CFG, the temporary GROUM of S now becomes a final GROUM, thus, it is added to the list A (line 14). If SN is not the return node, we need to consider all unexplored out going edges of SN. For each next node NN of SN, if NN is not in the set of all explored nodes of S, we copy all information of S into a new state S_c, set NN is the start node of S_c, and push S_c into F. The algorithm terminates when the stack F becomes empty. Our algorithm explores all the paths of CFG and only go through each loop at most one time. The list A now contains all GROUM of the method M. We next describe algorithm to build temporary GROUM.

Building temporary GROUM. Initially, before considering any instructions, we create an empty GROUM. For each parameter of the method we create a corresponding object node and add to the GROUM. When an instruction is considered (line 14), it creates a new action node and add that node to the current GROUM. After adding the new action node, we need to update edges. We add a control edge from the last inserted action node to the new action node. For data edges, we first need to add data edges from object nodes that are parameters of the instruction to the new action node. To do that, we maintain a map $M_{current}$ to store the current mapping between registers and object nodes. When an action node is created, we use $M_{current}$ to identify all object nodes that are used as parameters of the action, and we add data edges from those object nodes to the action node. Consider the instruction 000f: invoke-virtual v0, Ljava/io/BufferedReader;.readLine:() Ljava/lang/String;, it has an input parameter, which is the register v0, using $M_{current}$ we know that when we create an action node from the instruction, v0 is currenty holding a reference to a BufferedReader object node. We then add a data edge from the object node to the newly created action node. An instruction may return a value to a register. If it does not create a new object i.e. it returns a reference of an object has already been created before, then we identify the object node represent that object by $M_{current}$ and add a data edges from the action node to this node. Otherwise, we create a new object, add a data edge from the action node to the new object node and update mappings in $M_{current}$.

4.2 API Sequence Extractor

In this section, we describe how to extract API method sequences for one object or multiple objects from of a method using GROUM. For single object, sequence extractor scans through a GROUM to find object nodes associates with Android API objects. For each object node O_i we find all action nodes that have data edges connected with O_i and sort these action nodes by execution order. From the sorted action sequence, we only consider action nodes that represent API methods to get an API method sequence. For example, in Figure 9, for the first GROUM, there are three action nodes related to the object node BufferedReader and all of them are Android API methods, thus, the API method sequence associates the object node BufferedReader is (BufferedReader.init, BufferedReader.readLine, BufferedReader.close).

To extract action sequences that involve multiple API objects, we first identity usage-dependent object sets. For each action node that represents API methods, we collect all API object nodes that have data-dependent with it, then we form the set of object types correspond to those object nodes. In the first GROUM of Figure 9, there are two sets of usage-dependent objects: (FileReader, BufferedReader) has data-dependency with the action node BufferedReader.init, and (BuffedReader, String) has data-dependency with the action node BufferedReader.readLine.After collecting usage-dependent object sets, for each object set, we extract corresponding API method sequences using same technique for single object. For example, the API method sequence for the set (FileReader, BufferedReader) in the first GROUM is (FileReader.init, BufferedReader.init, BufferedReader.readLine, BufferedReader.close).

There should be only one method sequence for an object or a set of usage-dependent objects in each GROUM of a method. Thus, after extracting method sequences in all GROUMs of the method, we remove duplicate sequences and only keep distinctive sequences.

4.3 HAPI Learner

The HAPI learner uses a collection of API method sequences of an object (a set of usage-dependent objects) to learn the API usages model. In this section, we describe the training algorithm to estimate parameters of HAPI model from training data. We also present a method to choosing the number of hidden states.

4.3.1 Training Algorithm

The training algorithm aims to estimate HAPI's parameters $\lambda = (A, B, \pi)$ given a collection of API method sequences. In general, Baum-Welch algorithm is often used to estimate parameters of Hidden Markov Model [34]. In this paper, we present a modified version of Baum-Welch algorithm for the problem of learning API usages. The input of the algorithm is a collection of API method sequences. We observed that there are many method sequences are duplicated in the colllection. Thus, to save space and speed up the training algorithm, we store training data as a map, where each method sequence is mapped to the number of time it occurs in the collection. Initially, the parameters of a HAPI are assigned with random values. The main idea of the algorithm is to iteratively estimate parameters to maximize the likelihood function (the probability of generating data given model). The iterative process terminates when the estimated values of parameters converge.

Figure 10 shows the algorithm for training HAPI. Its input includes training data S and the number of hidden states K.

```
1   function TrainHAPI(TrainSet $S$, NHiddenStates $K$)
2     initialize $\lambda = (A, B, \pi)$ by random values
3     repeat
4       for $(Y_n = (y_1, y_2, ..., y_T), c_n) \in S$
5         $\alpha$ = Forward($\lambda, Y_n, T$)
6         $\beta$ = Backward($\lambda, Y_n, 1$)
7         for $i \in 1..K$: {compute state transition probabilities}
8           $\gamma_i^n(t) = \frac{\alpha_{i,t}\beta_{i,t}}{\sum_{j=1}^{K} \alpha_{j,t}\beta_{j,t}}$
9           for $j \in 1..K, t \in 1..T-1$:
10            $\xi_{ij}^n(t) = \frac{\alpha_{i,t} a_{ij} \beta_{j,t+1} b_j(y_{t+1})}{\sum_{k=1}^{K} \alpha_k(t)\beta_{k,t}}$
11      for $i \in 1..K$: {update model parameters}
12        $\pi_i^{(s+1)} = \frac{1}{D} \sum_{n=1}^{N} c_n \times \gamma_i^n(1)$
13        for $j \in 1..K$:
14          $a_{ij}^{(s+1)} = \frac{\sum_{n=1}^{N} c_n \times \sum_{t=1}^{T-1} \xi_{ij}^n(t)}{\sum_{n=1}^{N} c_n \times \sum_{t=1}^{T-1} \gamma_i^n(t)}$
15        for $v_m \in V$:
16          $b_i^{(s+1)}(v_m) = \frac{\sum_{n=1}^{N} c_n \times \sum_{t=1}^{T} 1_{y_t=v_m} \xi_i^n(t)}{\sum_{n=1}^{N} c_n \times \sum_{t=1}^{T} \gamma_i^n(t)}$
17    until convergence
18    return $\lambda$
19
20  function Forward(HAPI $\lambda$, APISequence Y, Position P)
21    $\alpha = \{\alpha_{i,t}\} \quad 1 \le i \le K, 1 \le t \le P$
22    for $i \in 1..K$: $\alpha_{i,1} = \pi_i b_i(y_1)$
23    for $i \in 1..K, t \in 2..P$:
24        $\alpha_{i,t} = b_i(y_t) \sum_{j=1}^{K} \alpha_{i,t-1} a_{ij}$
25    return $\alpha$
26
27  function Backward(HAPI $\lambda$, APISequence Y, Position P)
28    $\beta = \{\beta_{i,t}\} \quad 1 \le i \le K, P \le t \le T$
29    for $i \in 1..K$: $\beta_{i,T} = 1$
30    for $i \in 1..K, t \in T-1..P$:
31        $\beta_{i,t} = \sum_{j=1}^{K} \beta_{j,t+1} a_{ij} b_j(y_{t+1})$
32    return $\beta$
```

Figure 10: Training Algorithm

The parameters of the model are initialized randomly (line 1). Let us describe steps in the main loop of the algorithm: **Step 1**. Compute forward and backward probabilities. For each method sequence in training data we use dynamic programming to compute forward probabilities $\alpha_{i,t}$ (using Forward function) and backward probabilities $\beta_{i,t}$ (using Backward function) [34]. $\alpha_{i,t}$ is defined as the probability of seeing partial method sequence $y_1, y_2, ..., y_t$ and being in state q_i at time t given the model λ:

$$\alpha_{i,t} = P(y_1, y_2, ..., y_t, i_t = q_t|\lambda) \tag{1}$$

$\beta_i(t)$ is the probability of seeing the ending partial sequence $y_{t+1}, ..., y_T$ given state q_i at time t and the model λ:

$$\beta_{i,t} = P(y_{t+1}, ..., y_T|i_t = q_i, \lambda) \tag{2}$$

Step 2. Compute state probabilities γ and state transition probabilities ξ using forward and backward probabilities. $\gamma_i^n(t)$ is probability of being at state q_i at time t given method sequence Y_n and the model λ:

$$\gamma_i^n(t) = P(i_t = q_t|Y_n, \lambda) = \frac{\alpha_{t,i}\beta_{i,t}}{P(Y|\lambda)} \tag{3}$$

The state transition probability $\xi_{ij}^n(t)$ is the probability of being in state q_i at time t and making transition to state q_j at time $t+1$ given method sequence Y and the model λ:

$$\xi_{ij}^n(t) = P(i_t = q_i, i_{t+1} = q_j|Y_n, \lambda) \tag{4}$$

Step 3. Reestimate model parameters. In this step, the model parameters are estimated as expected values of probabilities that we computed in the previous step. $\gamma_i^n(t)$ is

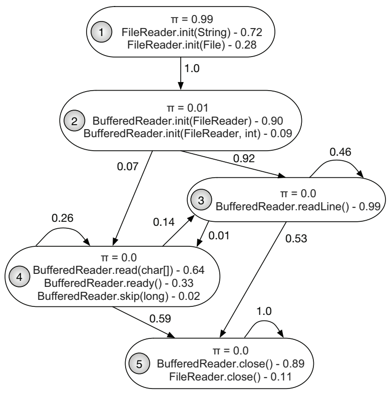
π = 0.99
1
FileReader.init(String) - 0.72
FileReader.init(File) - 0.28
1.0
π = 0.01
2
BufferedReader.init(FileReader) - 0.90
BufferedReader.init(FileReader, int) - 0.09
0.07
0.92
0.46
π = 0.0
3
BufferedReader.readLine() - 0.99
0.26
0.14
0.01
0.53
π = 0.0
4
BufferedReader.read(char[]) - 0.64
BufferedReader.ready() - 0.33
BufferedReader.skip(long) - 0.02
0.59
1.0
π = 0.0
5
BufferedReader.close() - 0.89
FileReader.close() - 0.11

Figure 11: HAPI for FileReader and BufferedReader objects

computed as the expected number of time state q_i is at time 1. $a_{ij}^{(s+1)}$ is estimated as the expected number of transitions from state q_i to state q_j compared to the expected total number of transitions from state q_i. $b_i^{(s+1)}(v_m)$ is the expected number of times the output method have been equal to v_m while in state i over the expected total number of times in state q_i. In the update equations, D is the total number of method sequences in the training set, and $1_{y_t=v_m}$ is the indicator function.

4.3.2 Choosing the number of hidden states

When training the HAPI model for an object (or a set of objects), we need to specify number of the hidden states as an input for the training algorithm, but we often do not know the how many states that the object has. Our idea in choosing the number of hidden states K is try to build models with different K and find a model that best describe new method sequences. This problem is equivalent to finding K that maximizes the probability of generating new data, i.e. likelihood function. In this method, we divide a training set into two sets, one is used to train models and one is used as validation data to optimize the number of hidden states. For each K in a range, we train a HMM model with K as the number of hidden states. Then, for each model, we compute the likelihood of validation data which is the probability of generating the validation data given the model. We then choose K of the model that maximize the likelihood of the validation set. Figure 11 shows graph representation the HAPI model for the pair of objects FileReader and BufferedReader as a result of HAPI learner component. To make it easier to display, we round probabilities and ignore probabilities that is less than 0.01.

4.4 API Usage Recommendation

In this section, we present an algorithm for recommending API method call using HAPI. The input of the algorithm is the HAPI model H of an object (set of objects) and the incomplete API method calls $Y = (y_1, ..., y_N)$ associated with the object (set of objects) in current editing code. The location T of Y is missing. The output of the suggestion algorithm is a ranked list of API methods that could be filled as the method call at position T. The idea of our algorithm is to place each API method as the method call of Y at location T and compute score of this assignment as the probability of generating the updated sequence (including

```
function NextAPICall(HAPI $\lambda$, APISequence $Y$, Location $T$)
  $R = \emptyset$ // a ranked list of candidates
  $\alpha$ = Forward($\lambda, Y, T-1$)
  $\beta$ = Backward($\lambda, Y, T+1$)
  for $v \in V$:
    for $i \in 1..K$:
      $\alpha_{i,T} = b_i(v) \sum_{j=1}^{K} \alpha_{j,T-1} a_{ij}$
      $\beta_{i,T} = \sum_{j=1}^{K} \beta_{j,T+1} a_{ij} b_j(v)$
      $score = \sum_{i=1}^{K} \alpha_{i,T}\beta_{i,T}$
      UpdateCandidateList($R, v, score$)
  return $R$
```

Figure 12: Algorithm for suggesting next method call

Table 1: Data Collection

Number of apps	207,603
Number of classes	32,080,884
Number of methods	59,636,164
Number of bytecode instructions	1,759,540,508
Space for storing .dex files	689.8 GB

the new API method) given the HAPI model. Then we add the API method with score to the ranked list of all candidates.

Figure 12 shows the algorithm. In the first part of our algorithm, we compute forward probabilities at position $T-1$ (using Forward function). We also compute backward probabilities at position $T+1$ (using Backward function). Then, we place each API method v as the method call of Y at position T and compute forward and backward probabilities at that position (line 6-8). The *score* of v is the probability of generating sequence $(y_1, ..., y_T = v, ..., y_N)$ is computed by summing all product of forward and backward probabilities:

$$P(Y, y_T = v|\lambda) = \sum_{i=1}^{K} P(Y, y_T = v, i_T = q_i|\lambda) = \sum_{i=1}^{K} \alpha_{i,T}\beta_{i,T}$$

The algorithm returns a ranked list of all the API method candidates with scores for suggestion.

5. EMPIRICAL EVALUATION

We conducted several experiments to study the run-time performance of SALAD and compare the accuracy of HAPI models and baseline models in recommending API method calls. All experiments are executed on a computer running 64-bit Ubuntu 14.04 with Intel Core i7 3.4Ghz CPU, 16GB RAM, and 1TB HDD storage.

5.1 Data Collection

Table 1 summarizes data collected for our experiments. In total, we downloaded and analyzed **207,603 apps** from 26 categories in Google Play Store. We only downloaded apps with the average rating of at least 3 (out of 5). This is based on the assumption that the high-rating apps would have high quality code, and thus, would contain API usages of high interest for learning.

Since Android mobile apps are distributed as .apk files, SALAD unpacked each .apk file and kept only its .dex file. The total storage space for the .dex files of all downloaded apps is around 700 GB. SALAD parsed those .dex files and obtained **32 millions classes**. It analyzed each class and looked for all methods in the class to build GROUM models.

Table 2: Extracting API method sequences

	Single object	Multiple object
Number of distinct object types (sets)	4,877	43,408
Total number of method sequences	195,692,154	143,678,399
Average number of method sequences	40,125	3,309
Average length of method sequences	3.32	7.72

Table 3: Training HAPI models

	Single object	Multiple object
Number of trained models	3,042	21,526
Average number of hidden states	14.07	15.65
Total training time	1h 20m	2h 30m
Total space to store trained models	30.3 MB	134.1 MB

Since an Android mobile app is self-contained, its .dex file contains bytecode of all external libraries it uses. That leads to the duplication of bytecode of shared libraries. Thus, SALAD maintains a dictionary of the analyzed methods, thus, is able to analyze each method only once. In the end, it analyzed **60 millions methods** which have in total nearly **1.8 billions** bytecode instructions.

5.2 Extracting API Method Sequences

SALAD built GROUM for each remaining method in the dataset and extracted API method sequences from those models. It extracted sequences for both single object usages and multiple object usages. Since we focus on learning Android API usages, only sequences involving classes and methods of Android application framework were extracted. Sequences with only one method call were disregarded.

Table 2 summarizes the extraction result. In total, SALAD has extracted nearly **195 millions method sequences** involving single object usages of more than **4,800** classes. There are more than **43,000** distinct usage-dependent object sets made from those types and SALAD has extracted over **143 millions method sequences** involving them. The running time and space is efficient. SALAD took totally 28 hours to build GROUMs and extract method sequences for 200 thousands apps, i.e. **0.5 seconds per app** on average. It needed less than 1.5 GB of working memory.

5.3 Training API Usage Models

Once all API method sequences are extracted and stored, SALAD trained HAPI models for each object type or usage-dependent object sets. For example, a HAPI is trained for the usages involving any MediaRecorder object and another HAPI is trained for the usages involving any two objects {FileReader, BufferedReader}. However, some HAPIs have too few sequences for training, making the training procedure unstable. Therefore, we do not train usage models for API objects with less than 10 method sequences. Overall, we discarded less than 237,170 sequences out of 350 millions.

The training process for the remaining models is summarized in Table 3. As seen in the table, SALAD is time- and space-efficient. It trained nearly 24,000 HAPI models in about 3 hours 50 minutes i.e. **0.55 second/model** on average. The total storage for all of them is of 160 MB, i.e. **7 KB/model** on average.

Table 2 and Table 3 show that SALAD can train HAPI models for 60% (3,042/4,877) of encountering API single object usages and 50% (21,526/43,408) of multiple object usages. However, untrained APIs are rarely used, each has less than 10 usages in 207,603 apps (59,636,164 methods).

5.4 API Usage Recommendation

We performed an experiment to measure the accuracy of HAPI in recommending API usages and compare to baseline models. In this experiment, we chose the task of recommending the next method call. That is, given an API method sequence and the model is expected to recommend the most probable next method call. This task has been used in the evaluation of prior approaches [28, 5].

In the experiment, we predicted and evaluated *all method calls* in every method sequence from the testing set. For a method call c_i at position i, we provided its $i-1$ prior method calls as the input and used the model to infer the top-k most probable next method calls $R_k = \{r_1, r_2, ..., r_k\}$. If c_i is in R_k, we consider it as a hit, i.e. an accurate top-k recommendation. The top-k accuracy is the ratio of the total hits over the total number of evaluated method calls.

5.4.1 Baseline models

We chose n-gram and recurrent neural network (RNN) trained for method call sequences as two baseline models for comparison due to the following reasons. First, both of them are statistical models, thus, is comparable to HAPI, which is also a statistical model. In addition, n-gram is widely used in recent research on code completion [14, 30]. More importantly, Raychev *et al.* recently evaluated RNN and n-gram and reported RNN as the best approach.

n-gram model is a simple statistical model for modeling sequences. An n-gram model learns all possible conditional probabilities $P(m_i|m_{i-n+1}...m_{i-1})$, where m_i is the current method call and $m_{i-n+1}...m_{i-1}$ is the sub-sequence of $n-1$ prior method calls. This is the probability that m_i occurs as the next method call of $m_{i-n+1}...m_{i-1}$. Using the chaining rule, we can use an n-gram model to compute the generating probability of any given method sequence $m_1...m_n$. In our experiment, we used a 3-gram model (i.e. the occurrence probability of a method call depends on its two previous calls) as used in prior work [35]. We also implemented Witten-Bell smoothing [43] technique for this model.

Recurrent neural network (RNN) is a class of neural networks for learning sequences. Like a HAPI, a RNN can be trained with a collection of method sequences and then is able to compute the probability of the next method call for any given method sequence. In other words, RNN can compute all conditional probabilities $P(m_i|m_1...m_{i-1})$ for any given a method sequence $m_1...m_n$. To do that, it maintains a context vector c_i represents current context of sub-sequence up to $m_1...m_{i-1}$. A function f is learned from data to compute the context vector at position i, $c_i = f(m_i, c_{i-1})$ given the current method call m_i and previous context c_{i-1} while another function g is learned to compute the probability of the next call m_{i+1}, $P(m_{i+1}|m_1...m_i) = g(c_i)$ given the current context c_i. In our experiment, we used a publicly available implementation of RNN[1] with the number of hidden states of 40 which is also used in prior work [35].

5.4.2 Experiment results

Our experiment is a 5-fold cross validation. That is, for each object type or object set in the experiment data, we divided its method sequences into five equal folds. HAPI and two baseline models are trained in four folds and tested in the remaining folds (i.e. they are trained and tested in the same data). We repeated this process five times, each for an individual fold as test data and computed the average accuracy of each model in five iterations as its final result. We chose 5-fold cross validation over the popular 10-fold cross validation to reduce experiment time because RNN is very time-consuming.

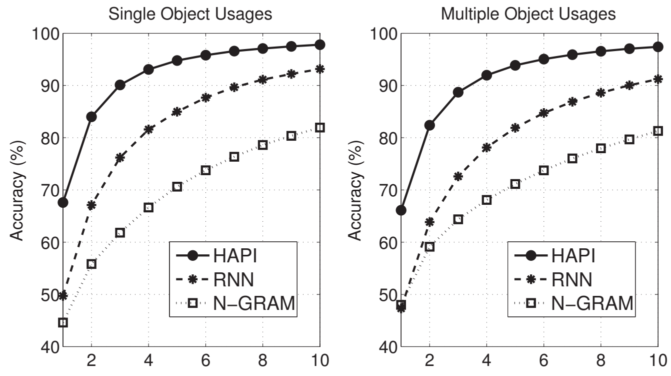

Figure 13: Accuracy of next method call recommendation

Figure 13 shows the experiment results for method sequences extracted from bytecode of 200 thousand apps downloaded from Google Play. As seen in the charts, all three models have consistent accuracy for single and multiple object usages. More importantly, HAPI can recommend API usages with very high levels of accuracy. For example, for single object usage, it has a top-3 accuracy of 90% and a top-10 accuracy of 98%. In addition, HAPI significantly outperforms RNN and n-gram models. For example, the corresponding top-3 and top-10 accuracy of n-gram model are 62% (28% lower) and 82% (16% lower). The top-3 and top-10 accuracy of RNN model are 76% (14% lower) and 93% (5% lower). On average, HAPI has 11% improvement over RNN and 21.5% improvement over n-gram.

5.4.3 Discussions

Occam's razor principle could explain HAPI's improvement over n-gram and RNN. To model usages involving M methods, a HAPI with K hidden states uses $K + K^2 + KM$ parameters, while an n-gram uses M^n parameters (i.e. all possible sub-sequences of n method calls), and a RNN of H hidden nodes uses $MH + H^2 + HM$ parameters. Since K is often much smaller than H and M, HAPI has less parameters thus easier to train and generalize.

6. RELATED WORK

There exist several works that proposed statistical models for learning API usages. The most similar research to SALAD is SLANG [35]. SLANG uses n-gram and recurrent neural networks (RNN) to learn API usage patterns per object which are used to predict and suggest next API calls. Compared with these two models, HAPI has less parameters thus easier to train and generalize. Nguyen et al. [29] introduced GraLan, a graph-based statistical language model that learns common API usage (sub)graphs from a source code corpus and computes the probabilities of generating

[1]http://www.rnnlm.org

Table 4: Experiment settings in recent research about statistical language models for source code

Approach	Model	Code token	Code form	Training	Testing
SALAD	3-gram, RNN, HAPI	Method call (Android)	Bytecode	207,603 apps/59,636,164 methods	5-fold cross validation
SLANG [35]	3-gram, RNN	Method call (Android)	Source code	3,090,194 methods	20 pre-selected examples
GraLan [29]	9-gram	Method call (Java)	Source code	1,000 projects	5 independent projects
SLAMC [30]	3-gram	Semantic token	Source code	9 (Java) + 9 (C#) projects	10-fold cross validation
Tu et. al [39]	3-gram (cache)	Lexical token	Source code	9 (Java) + 9 (Python) projects	10-fold cross validation
Hindle et. al [14]	3-gram	Lexical token	Source code	5 projects	200 held-out source files

new usage graphs given the observed (sub)graphs. Although graphs are better than sequences in capturing context information, the number of sub-graphs can grow exponentially. That means, training sequence-based models would be more time- and space-efficient. In our experiment, we do not chose GraLan as a baseline to compare with HAPI because as a graph-based statistical model for API usages, it is incompatible with the sequence-based models. (GraLan models graphs, not method sequences, thus, cannot operate in the same experiment settings with HAPI, n-gram, and RNN). In addition, it is potentially unscalable and likely less accurate. It is reported in [29] that training GraLan on 1,000 projects needs 20 hours and 4.5GB [29] (while we have 207,603 apps and its top-3 accuracy is just of 63% (while top-3 accuracy of HAPI is 90%).

Statistical models for capturing rules and patterns in source code become a hot research topic in software engineering in the recent years. Hassan et al. [12] indicated "natural" software analytics based on statistical modeling will become one of the most important of aspects of software analytics. Hindle et al. [14] showes that source code is repetitive and predictable like natural language and they adopted n-gram model on lexical tokens to suggest the next token. SLAMC [30] represents code by semantic tokens, i.e. annotations of data types, method/field signatures, etc. rather than lexical tokens. SLAMC combines n-gram modeling of consecutive semantic tokens, topic modeling of the whole code corpus, and bi-gram of related API functions. Tu et al. [39] exploited the localness of source code. White et al. [42] proposed deep learning approach modeling source code. Allamanis and Sutton [3] trains n-gram language model a giga-token source code corpus. NATURALIZE [2] use n-gram language model to learns the style of a codebase and suggest natural identifier names and formatting conventions. Jacob et al. [21] uses n-gram model to learn code templates. Hidden Markov Model has been used infer the next token from user-provided abbreviations [11] and detect coded information islands, such as source code, stack traces, and patches, from free tex [8]. Maddison et al. [25] proposed tree-based generative models for source code. Hsiao et al. [17] learns n-gram language model on program dependence graph and uses the model for finding plagiarized code pairs. Table 4 summarizes the differences of SALAD and some of those approaches. It suggests that the differences between our results and those published studies might result from differences in code token types and experiment data.

Pattern mining approaches represent usage patterns using various data structure such as sequences, sets, trees, and graphs. JADET [41] extracted a usage model as a set of partial order pairs of method calls. MAPO [45] mined frequent API call sequences and suggests associated code examples. Wang et al. [40] mines succinct and high-coverage API usage patterns from source code. Acharya et al. [1] proposed an approach to mine partial orders among APIs. Buse and Weimer [7] propose an automatic technique for mining synthesizing succinct and representative human-readable API examples. Other techniques includes mining associate rules [24], item sets [6], subgraphs [31], [9], code idioms [4], topic modeling [27], etc.

One application of usage patterns mined from existing code is to support code completion. Grapacc [28] mines and stores API usage patterns as graphs and suggest these graphs in current editing code. Bruch et al. proposed three approaches for code completion. First, FreqCCS recommends the most frequently used method call. Second, ArCCS mines associate rules and suggest methods that often occur together. Finally, a best matching neighbors code completion technique that makes used k-nearest-neighbor algorithm. SLANG [35] uses n-gram to suggest the next API call based on a window of $n-1$ previous methods. Precise [44] mines existing code bases and builds a parameter usage database. Upon request, it queries the database and recommends API parameters. Graphite [33] allows library developers to introduce interactive and highly-specialized code generation interfaces that could interact with users and generates appropriate source code.

Other approaches have been proposed to improve code completion tasks. Robbes et al. [36] improves code completion with program history. They measure the accuracy of replaying entire change history of programs with completion engine and gather information for improvement. In [16], the authors found that ranking method calls by frequency of past usages is effective and propose new strategies for organizing APIs in the code completion proposals. Hill and Rideout [13] matches the code fragment under editing with small similar-structure code segments that frequently exist in large software projects. The authors of [26] and [37] use API documentation to suggest source code examples to developers. Holmes and Murphy [15] describe an approach for locating relevant code examples based on heuristically matching with the structure of the code under editing.

7. CONCLUSIONS

We propose a statistical approach to learn API usages from bytecode of Android mobile apps. The key component of our approach is HAPI, a statistical, generative model of API usages. We developed three algorithms to extract method sequences from apps' bytecode, to train HAPI based on those method sequences, and to recommend method calls in code completion engines using the trained HAPIs. Our empirical evaluation on a very large dataset of more than 200 thousands apps indicates that our approach can effectively learn API usages from apps' bytecode and provide API recommendations with high levels of accuracy and outperform the baseline models.

8. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the ACM SigSoft Symposium on Foundations of Software Engineering*. ACM âĂŞ Association for Computing Machinery, November 2014.

[3] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 207–216, May 2013.

[4] M. Allamanis and C. Sutton. Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM.

[5] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. CSCC: simple, efficient, context sensitive code completion. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 71–80, 2014.

[6] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[7] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press.

[8] L. Cerulo, M. Ceccarelli, M. Di Penta, and G. Canfora. A hidden markov model to detect coded information islands in free text. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 157–166, Sept 2013.

[9] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *Software Engineering, IEEE Transactions on*, 34(5):579–596, Sept 2008.

[10] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[11] S. Han, D. R. Wallace, and R. C. Miller. Code completion from abbreviated input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 332–343, Washington, DC, USA, 2009. IEEE Computer Society.

[12] A. E. Hassan, A. Hindle, P. Runeson, M. Shepperd, P. T. Devanbu, and S. Kim. Roundtable: What's next in software analytics. *IEEE Software*, 30(4):53–56, 2013.

[13] R. Hill and J. Rideout. Automatic method completion. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 228–235, Washington, DC, USA, 2004. IEEE Computer Society.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[15] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[16] D. Hou and D. M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. In *ICSM*, pages 233–242. IEEE, 2011.

[17] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, pages 49–65, New York, NY, USA, 2014. ACM.

[18] http://developer.android.com.

[19] https://code.google.com/p/smali/.

[20] https://github.com/Akdeniz/google-play crawler.

[21] F. Jacob and R. Tairas. Code template inference using language models. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 104:1–104:6, New York, NY, USA, 2010. ACM.

[22] J. Kim, S. Lee, S. won Hwang, and S. Kim. Towards an intelligent code search engine. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.

[23] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, New York, NY, USA, 2013. ACM.

[24] D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 109–118, Washington, DC, USA, 2008. IEEE Computer Society.

[25] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. In *The 31st International Conference on Machine Learning (ICML)*, June 2014.

[26] C. McMillan, D. Poshyvanyk, and M. Grechanik. Recommending source code examples via api call usages and documentation. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 21–25, New York, NY, USA, 2010. ACM.

[27] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk,

M. Grechanik, C. McMillan, and M. Gethers. Export: Detecting and visualizing api usages in large source code repositories. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 646–651, Nov 2013.

[28] A. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. Nguyen, J. Al-Kofahi, and T. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 69–79, June 2012.

[29] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press.

[30] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM.

[31] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[32] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen. Recommending api usages for mobile apps with hidden markov model. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 795–800, Nov 2015.

[33] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press.

[34] L. R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSp Magazine*, 1986.

[35] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM.

[36] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[37] S. Subramanian, L. Inozemtseva, and R. Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM.

[38] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 283–297, Riverton, NJ, USA, 2013. IBM Corp.

[39] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014. ACM.

[40] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 319–328, May 2013.

[41] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[42] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 334–345, Piscataway, NJ, USA, 2015. IEEE Press.

[43] I. H. Witten and T. Bell. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *Information Theory, IEEE Transactions on*, 37(4):1085–1094, Jul 1991.

[44] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press.

[45] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 318–343, Berlin, Heidelberg, 2009. Springer-Verlag.