2015 30th IEEE/ACM International Conference on Automated Software Engineering

Recommending API Usages for Mobile Apps with Hidden Markov Model

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 apps often rely heavily on standard API frameworks and libraries. However, learning to use those APIs is often challenging due to the fast-changing nature of API frameworks and the insufficiency of documentation and code examples. This paper introduces DroidAssist, a recommendation tool for API usages of Android mobile apps. The core of DroidAssist is HAPI, a statistical, generative model of API usages based on Hidden Markov Model. With HAPIs trained from existing mobile apps, DroidAssist could perform code completion for method calls. It can also check existing call sequences to detect and repair suspicious (i.e. unpopular) API usages.**

***Keywords*—Statistical code completion, API usage.**

I. Introduction

Due to the fierce competition, mobile apps often have very short time-to-market and upgrade cycles. Thus, to reduce development time, app developers extensively reuse API application frameworks and libraries (e.g. Android, iOS frameworks, Java APIs). For example, an Android app might have up to 42% of its external dependencies to Android APIs and 68% to Java APIs [1].

Learning and using APIs is challenging due to several reasons. First, a framework often consists large numbers of API functions and types. For example, Android application framework contains over 3,400 classes and 35,000 methods, clustered in more than 250 packages [2]. Moreover, typical API usage scenarios often include several API elements and follow special rules, e.g. for pre- and post-conditions or for control and data flows [3]–[5]. Unfortunately, API documentation is often insufficient. For example, the Javadoc of a class often contains only descriptions of its methods and rarely has code examples on the usage of its objects and methods [6]. Documentation and code examples for API usages involving several objects are often non-existed.

The situation is even more difficult for learning APIs of mobile frameworks. First, due to the fast development of mobile devices and the strong competition between vendors, those frameworks are often upgraded quickly and include very large changes. For example, 17 major versions of Android framework have been released within five years, making nearly 100,000 changes to its API methods [2]. More severely, mobile apps are often closed-source, i.e. their source code is not publicly available. Thus, finding and learning code examples from existing mobile apps projects would be difficult.

To address this problem, we develop DroidAssist, a recommending tool for API usages. When a developer is writing code, DroidAssist can analyze the code being written and recommend (and fill-in on request) the next or missing API method calls. To help the developer makes more effective choices, those calls are ranked based on their likelihoods of appearance in the existing code context. DroidAssist can also detect suspicious API usages in existing code (i.e. ones rarely/unlikely to be used) and repair them with more probable usages. More details will be discussed in the next sections. The tool and video demonstration are available at our website [1].

II. Usage Scenarios

DroidAssist is released as a plugin of Android Studio [7], the standard IDE for Android apps development. After installation, it is incorporated with Android Studio and users can invoke it directly from the current editing view (for method call recommendation) or via the menu (for method sequence validation). This section presents its two usage examples.

A. Recommending Next Method Call

Assume that the developer wants to write code for a database transaction. She has created a database query that return a `Cursor` object and made a call to begin the transaction. However, she forgets how to use the returned `Cursor` object properly. She invokes the built-in code completion engine, but it just lists all methods that can be called on the `Cursor`, thus does not help her to make an appropriate selection.

Now, with DroidAssist, the developer will have better recommendations. Figure 1 shows a screenshot of Android Studio with DroidAssist invoked (via the keystroke `Ctrl + Shift + Space`) for the current editing code. As seen, DroidAssist displays a ranked list of methods that can be called for the `Cursor` object. Each method has a score represents the probability of how likely it might be called in the given current context of the object and other interacting objects. In this example, method `moveToFirst` has the highest score of 30.52%. If the developer chooses it, it will be filled in the current position in the editor.

DroidAssist uses code context including the current method calls to infer and recommend the next call. For example, in Figure 2, object `db` already has three method calls: `beginTransaction`, `insert`, and `setTransactionSuccessful`. Thus, DroidAssist predicts that the next method call will be `endTransaction` with a probability of 61.38%, which is the highest among all available methods for this object.

[1] http://useal.cs.usu.edu/droidassist/

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.109

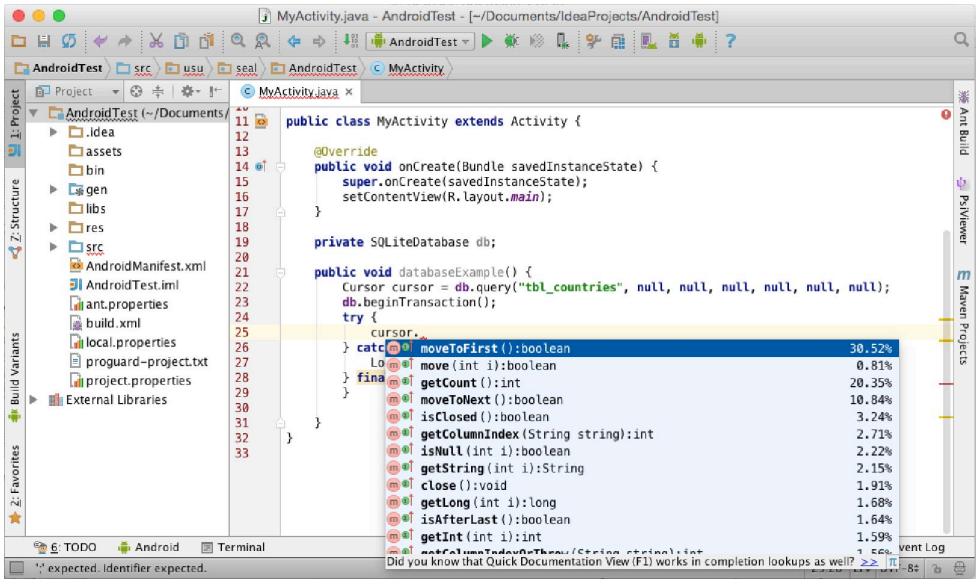

Fig. 1: Method Call Recommendation by DroidAssist

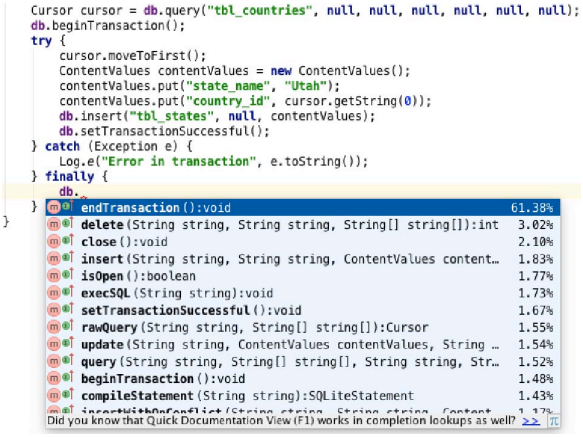

Fig. 2: Call recommendation for object db

DroidAssist recommendations are more accurate with more context information. For example, if the method sequence for object db has only two calls: beginTransaction and insert, setTransactionSuccessful has a probability of 15.11%. However, if endTransaction is already added, probability of setTransactionSuccessful increases dramatically to 65.65% (see Figure 3).

B. Analyzing Method Sequence

DroidAssist can analyze a given method sequence in existing code report. If it is a suspicious API usages (i.e. is rarely used or unlikely to be used), DroidAssist can offer fixes with more probable method sequences. Figure 4 shows a code example involving a MediaRecorder object. After writing code, the developer wants to check if his usage is acceptable. To invoke DroidAssist for that task, she moves to a line of code containing mediaRecorder variable, opens Analyze menu and selects Analyze API Call Sequence.

DroidAssist then will analyze the API usage for that object. Figure 5 shows the analyzed result. The left of the dialog shows the original method sequence. The right of the dialog shows the suggestions for repair. If DroidAssist detects a suspicious method sequence, it will suggest three actions: replace, add, or delete a method call, to make the usage more probable. In the example, DroidAssist detects that calling setAudioChannels at the beginning might not be a proper usage. It recommends to replace that method by setAudioSource. If the user choose option Replace and press Apply, DroidAssist will repair the API call sequence in the code editor by replacing setAudioChannels with setAudioSource.

III. Design and Implementation

This section briefly discusses the key points on design and implementation of DroidAssist. We will start with HAPI (*"Hidden Markov model of API usages"*), the statistical, generative model used by DroidAssist to (compactly) represent the API usages. Then we introduce the training processes for HAPI from the API usages in existing software code. Finally, we discuss how DroidAssist uses the trained HAPIs to perform its two aforementioned functions.

A. HAPI - Hidden Markov Model of API Usages

Let us introduce HAPI via an example. Figure 6 illustrates the usage of a MediaRecorder object in Android API as a state diagram reproduced from Android Developer website[2]. As seen in the figure, 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 the change of its state when a method (drawn as the label of the edge) is called.

We learn from the 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, it is in *Initial* state. If setAudioSource

[2]http://developer.android.com/reference/android/media/MediaRecorder.html

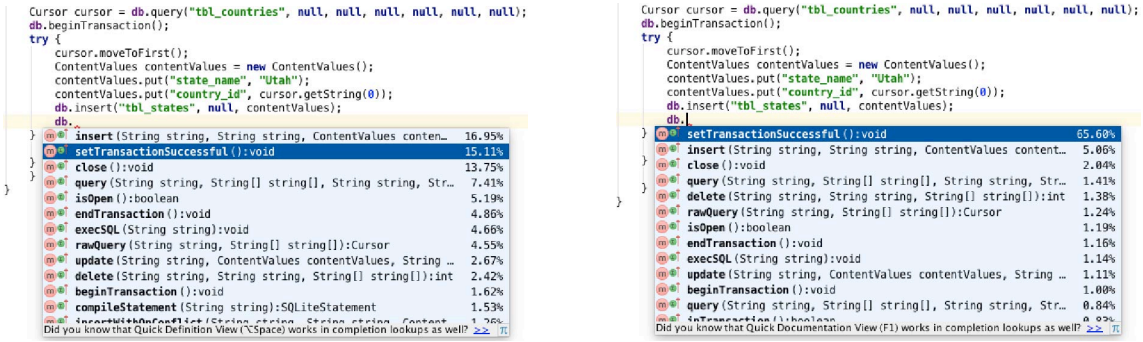

Fig. 3: API call recommendation before and after calling endTransaction method

```
public void mediaRecorderExample(String fileName) {
  mediaRecorder.setAudioChannels(2);
  mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
  mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
  mediaRecorder.setOutputFile(fileName);
  try {
    mediaRecorder.prepare();
  } catch (IOException e) {
    e.printStackTrace();
  }
  ...
}
```

Fig. 4: Source code examples of using MediaRecorder objects

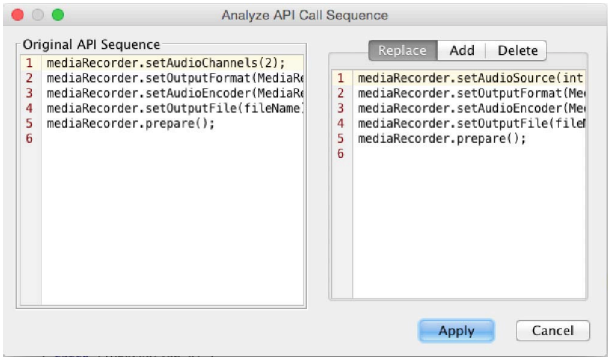

Fig. 5: Repair suggestion for suspicious usage

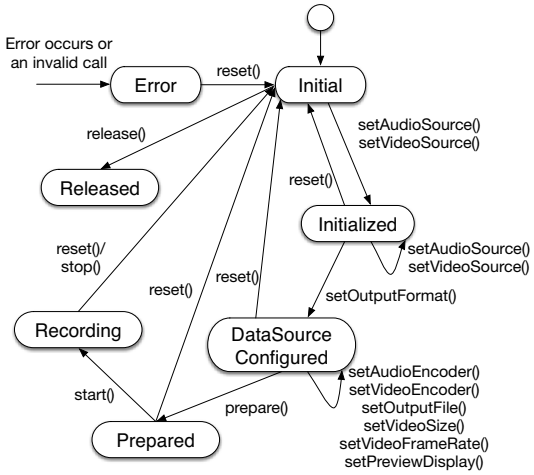

Fig. 6: State Diagram of MediaRecorder object

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

State diagrams are useful to understand API usages. For example, we could infer from Figure 6 the following method call sequence to perform an audio recording task.

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

Unfortunately, API documentation often does not provide state diagrams for most of API objects. In addition, current code completion engines like the one built-in in Android Studio do not utilize state diagrams in their recommendations.

HAPI is designed to take the usefulness of state diagrams and address their aforementioned weakness. In essential, a HAPI is a probabilistic state diagram, i.e. its nodes and edges associate with probabilities of invocations and transitions. More importantly, its structure and probabilistic parameters can be learned, i.e. estimated from data.

Figure 7 illustrates the HAPI learned by DroidAssist on the usages of MediaRecorder. As seen in the figure, each node of the HAPI represents a state of the object. But different from the state diagram in Figure 6, 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 of calling a particular method and another distribution specifying the probability of changing to another state. (It should be noted that, to simplify the drawing, we only show methods and transition edges with significant probabilities in the figure.)

As modeled by the HAPI, a MediaRecorder object starts in state (1) with probability of 83%. In this state, method setAudioSource is called with a probability of 82% and the

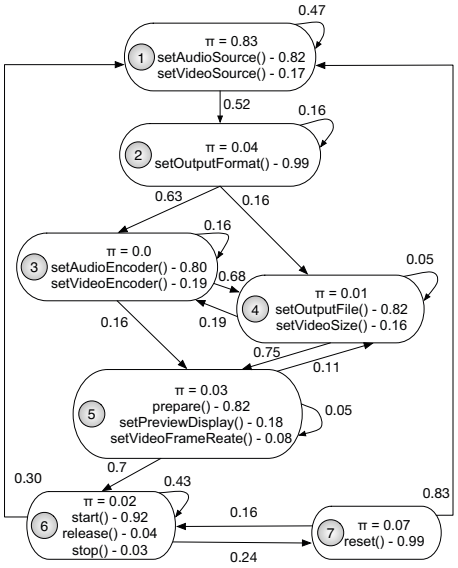

Fig. 7: The HAPI model to represent usage patterns of MediaRecorder object

object can change to state (2) with a probability of 52%. In that state, setOutputFormat is called with a probability of 99%. Technically, a HAPI is a Hidden Markov Model [8] on which we can 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.

B. Design Overview

Figure 8 shows the design overview of DroidAssist. It has five major components including two modules to extract API usages (represented as graph-based object usage models - GROUM [3]) from source code and bytecode, a module to extract method call sequences from those usage models, a module to train HAPIs from those sequences, and two modules that use the trained HAPIs to recommend next method calls or check/repair an existing method sequence. Let us describe those modules in more details.

1) GROUM Builder: DroidAssist uses GROUM (Graph-based Object Usage Model) [3], developed by the fourth author and his colleagues to represent the raw API usages in source code and bytecode. Thus, to collect training data, DroidAssist has a module to extract GROUMs from bytecode of existing Android apps and a similar module to extract GROUMs from the code being written, which are used for its two tasks of method call recommendation and validation. Figure 9 shows an example of GROUM built for the code snippet in Figure 2.

As seen, there are three kind of nodes in GROUM: object nodes represent objects and variables, method nodes represent method invocations, and control nodes represent control statements. There are two kinds of edges: data edges represent data dependency between object nodes and method nodes, while control edges represent temporal order between

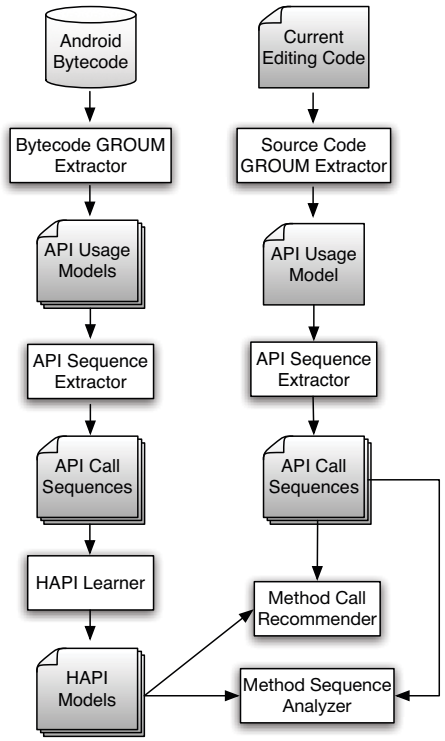

Fig. 8: Design Overview of DroidAssist

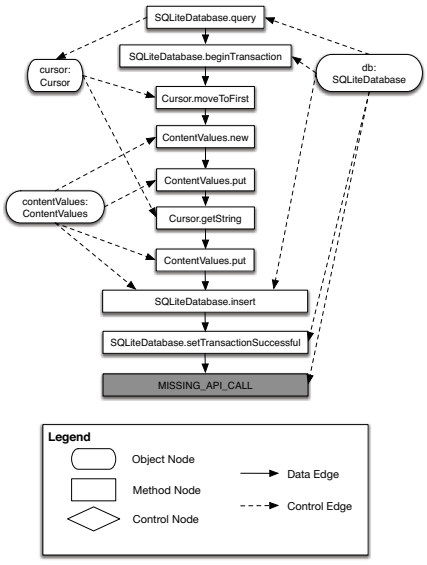

Fig. 9: A GROUM example

method/control nodes. Due to space constraints, we do not present the GROUM extraction algorithms here. They can be found at [3] and [9].

2) API Method Sequence Extractor: Because HAPI is designed based on Hidden Markov Model, it works on sequences, not graphs. Thus, DroidAssist has a module named API Sequence Extractor to extract sequences of method calls

from the GROUMs. For a given GROUM, this module simply traverses all its paths and extracts the method calls along those paths. However, to ensure the extracted method sequences are meaningful, it only keeps the sequences involving the same API objects (i.e. have data dependency). More details can be found at [9].

3) HAPI Learner: This module is responsible for training HAPI models from the extracted method sequences. It first sorts those method sequences by the involving API objects. For example, all method sequences involving the usages of a single MediaRecorder object are grouped together and then will be used to train the HAPI representing the usage of a single MediaRecorder object. Sequences involving the usages of a Scanner and a StringBuilder object are grouped and used to train the HAPI for those two objects.

DroidAssist uses a modified version of Baum-Welch algorithm to train a HAPI [9]. In the training process, it figures out the optimal number of internal state of the HAPI and estimates all probabilities involving the HAPI's states (i.e. how likely a state is selected as the starting state, how likely the object changes from a state to another) and method calls (i.e. how likely a method is called when the object is in a state). Please refer to [9] for more details of this algorithm.

4) Method Call Recommender: This module provides the recommendations of the next method call for the currently editing code (and performs the completion task if requested). Its input is a collection of method sequences provided by API Sequence Extractor along with the position of the missing call. It places every available method m at the missing position and computes the probability of the newly created sequence based on the HAPI of the API objects involving that sequence. It then combines those probabilities into a final score for method m. All methods then are ranked by those scores and proposed to the developer, as seen in Figure 2.

5) Method Sequence Analyzer: This module evaluates an API call sequence. Its input is a method sequence for one or more API objects. This module uses the corresponding HAPI to compute the probability of this sequence. If that probability is smaller than a threshold, the sequence is considered to be suspicious. If that happens, the Method Sequence Analyzer will explore different modifications on the sequence (i.e. by adding/replacing/removing a method call) to find more probable sequences and offers them for repair.

The threshold is computed based on the distribution of probabilities of all unique method sequences used for training the HAPI. Based on our experiment, we currently use the 75% percentile as the threshold. That means, if the HAPI is trained with 100 unique sequences, and 75 of them have probabilities smaller than 0.1, this value is chosen as the threshold.

IV. Related Work

Several tools for API usage recommendation have been developed. Grapacc [4] is the most similar to DroidAssist. However, it uses a graph-based pattern approach. That is, it mines, stores, and recommends API usage patterns as frequent graph-based models (GROUM [3]). As such, it can recommend only popular API usages. In contrast, DroidAssist uses a sequence-based, generative approach. Although it uses GROUM to extract API usages, it uses method sequences to train the HAPIs. Because HAPI is a generative model, it can model and recommend method sequences that are non-frequent or even unseen in training data.

SLANG [10] uses n-grams to model API usages and suggest the next API call. Thus, the context of a method call is a window of $n-1$ previous calls. In contrast, HAPI in DroidAssist can model method sequences of much longer length, i.e. can capture more context information. GraLan [11] models API usage by a graph-based statistical language model. 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.

Bruch et al. [12] proposed three techniques for code completion: 1) FreqCCS recommends the most frequently used method call, 2) ArCCS mines associate rules and suggests methods that often occur together and 3) BMN uses k-nearest-neighbor algorithm to find the best matched method sequences. That means, the context for a method call is an empty set, an another call, or un-ordered set of method calls. In comparison, context in DroidAssist contains much longer call sequences, thus provide more information.

Precise [13] mines existing code bases and builds a parameter usage database. Upon request, it queries the database and recommends API parameters. Graphite [14] 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. [15] improves code completion with program history. Hou and Pletcher [16] found that ranking method calls by frequency of past use is effective and propose new strategies for organizing APIs in the code completion pop up. Hill and Rideout [17] matches the fragment under editing with small similar-structure code segments that frequently exist in large software projects. McMillan et al. [18] and Subramanian et al. [19] use API documentation to suggest source code examples to developers. Holmes and Murphy [20] describe an approach for locating relevant code examples based on heuristically matching with the structure of the code under editing. MAPO [21] is a code recommendation tool which mines API usage patterns and recommends associated code examples. Codota [22] is a code example recommendation plugin on Android Studio. The function of Codota is similar to MAPO as it only recommends Android code examples.

V. Conclusion

This paper introduces DroidAssist, a recommendation tool for API usages of Android mobile apps. DroidAssist uses HAPI, a statistical, generative model designed based on Hidden Markov Model to model API usages. It can extract API usages (as graphs and sequences of method calls) from source code and bytecode to train its HAPI models and build the code context for its recommendation tasks. Using the HAPIs trained from existing apps, DroidAssist can perform code completion for method calls. It can also check existing call sequences to detect and repair suspicious (i.e. unpopular) API usages.

REFERENCES

[1] 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*, ser. CASCON '13. Riverton, NJ, USA: IBM Corp., 2013, pp. 283–297. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555523.2555553

[2] 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*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 477–487. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491428

[3] 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*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 383–392. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595767

[4] 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*, June 2012, pp. 69–79.

[5] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 422–431. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486844

[6] J. Kim, S. Lee, S. won Hwang, and S. Kim, "Towards an intelligent code search engine." in *AAAI*, M. Fox and D. Poole, Eds. AAAI Press, 2010. [Online]. Available: http://dblp.uni-trier.de/db/conf/aaai/aaai2010.html#KimLHK10

[7] Android studio. [Online]. Available: http://developer.android.com/tools/studio/index.html

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

[9] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning api usages from bytecode: A statistical approach," *arXiv:1507.07306 [cs.SE]*, 2015.

[10] 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*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[11] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015.

[12] 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*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 213–222. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595728

[13] 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*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 826–836. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337321

[14] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 859–869. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337324

[15] 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*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 317–326. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.42

[16] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion." in *ICSM*. IEEE, 2011, pp. 233–242. [Online]. Available: http://dblp.uni-trier.de/db/conf/icsm/icsm2011.html#HouP11

[17] R. Hill and J. Rideout, "Automatic method completion," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ser. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 228–235. [Online]. Available: http://dx.doi.org/10.1109/ASE.2004.19

[18] 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*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 21–25. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808925

[19] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 643–652. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568313

[20] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 117–125. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062491

[21] 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*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_15

[22] codota. [Online]. Available: http://www.codota.com