# Generating API Call Rules from Version History and Stack Overflow Posts

SHAMS AZAD, PETER C. RIGBY, and LATIFA GUERROUJ, Concordia University

Researchers have shown that related functions can be mined from groupings of functions found in the version history of a system. Our first contribution is to expand this approach to a community of applications and set of similar applications. Android developers use a set of application programming interface (API) calls when creating apps. These API calls are used in similar ways across multiple applications. By clustering co-changing API calls used by 230 Android apps across 12k versions, we are able to predict the API calls that individual app developers will use with an average precision of 75% and recall of 22%. When we make predictions from the same category of app, such as Finance, we attain precision and recall of 81% and 28%, respectively.

Our second contribution can be characterized as "programmers who discussed these functions were also interested in these functions." Informal discussions on Stack Overflow provide a rich source of information about related API calls as developers provide solutions to common problems. By grouping API calls contained in each positively voted answer posts, we are able to create rules that predict the calls that app developers will use in their own apps with an average precision of 66% and recall of 13%.

For comparison purposes, we developed a baseline by clustering co-changing API calls for each individual app and generated association rules from them. The baseline predicts API calls used by app developers with a precision and recall of 36% and 23%, respectively.

CCS Concepts: • **Software and its engineering** → *Software evolution*; *Maintaining software*; *Search-based software engineering*

Additional Key Words and Phrases: API method calls, community of applications, version history, informal documentation, Stack Overflow, association rule mining

**ACM Reference Format:**
Shams Azad, Peter C. Rigby, and Latifa Guerrouj. 2017. Generating API call rules from version history and Stack Overflow posts. ACM Trans. Softw. Eng. Methodol. 25, 4, Article 29 (January 2017), 22 pages.
DOI: http://dx.doi.org/10.1145/2990497

## 1. INTRODUCTION

Software developers use application programming interfaces (APIs) to access the functionality of libraries and frameworks. By using an API, developers reduce the cost of development and increase the quality of their application. Learning APIs can be difficult because developers often struggle to understand the structure of the API and the usage patterns of API calls [Robillard and DeLine 2011]. Furthermore, APIs evolve and developers must learn the new method calls to achieve a desired behavior.

Authors' addresses: S. Azad, SAP, 111 Rue duke, Suite 9000, Montreal, Quebec, Canada, H3C2M1; email: shams.abubkar.azad@sap.com; P. C. Rigby, Concordia University, Department of Computer Science & Software Engineering, Sir George William Campus, 1455 De Maisonneuve Blvd. W., Montreal, Quebec, Canada, H3G 1M8; email: peter.rigby@concordia.ca; L. Guerrouj, Department of Software Engineering and Information Technologies, École de Technologie Supérieure, 1100 Rue Notre-Dame Ouest, Montreal, Quebec, Canada, H3C 1K3; email: Latifa.Guerrouj@etsmtl.ca.

DOI: http://dx.doi.org/10.1145/2990497

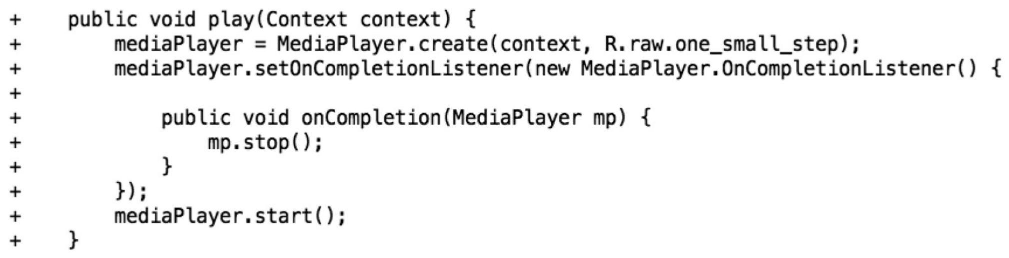
To help developers learn APIs, researchers have developed API usage patterns [Michail 1999; Robillard et al. 2013]. These patterns can be used in a wide variety of settings, from suggesting autocompletions in an IDE to generating new documentation for each API call. However, evaluations of these usage patterns can be difficult, as there is no benchmark against which to compare. Many researchers have used manual evaluations, and others have determined how often particular patterns are used in a set of applications. The former approach suffers from the usual biases in manual evaluations and user studies, whereas the latter simply represents how often a pattern is used. In both cases, it is difficult to compare among studies [Robillard et al. 2013]. To overcome these evaluation difficulties, we create a benchmark of API call rules based on the source code version history of 230 Android apps with 12k versions. We apply association rule mining on sets of API calls that change together in the version history of these apps to create our test set benchmark.

Our contribution is to use this test set to evaluate the rules generated from two separate data sources, the second of which is novel. Research approach 1 (RA 1) uses source code changes. The novel research approach 2 (RA 2) groups API calls that co-occur in Stack Overflow posts. In total, we have seven subapproaches. In RA 1.1, we follow Zimmerman et al. [2005] and create a baseline where we consider only the API changes made to individual apps. In RA 1.2, we generate rules from a community of apps. Mining rules from a community of software systems, such as Java applications, is the most common approach in the literature. We suspect that the rules generated from a large community may be too diverse, so in RA 1.3 we make a new contribution by suggesting changes that come from a subset of similar apps, such as apps restricted to the Finance category.

For RA 2, we assume that the API calls discussed together in a post will be helpful in suggesting changes that developers may want to use in their apps. In RA 1.1, we generate rules from all co-occurring API calls in positively voted answer posts. In RA 1.2, we consider only those API calls that are part of code snippets. Code snippets contain setup code that often dilute the salience of the co-occurrences between API calls, so in RA 1.3 we consider only those API calls that are part of a freeform text discussion—that is, not part of a code snippet.

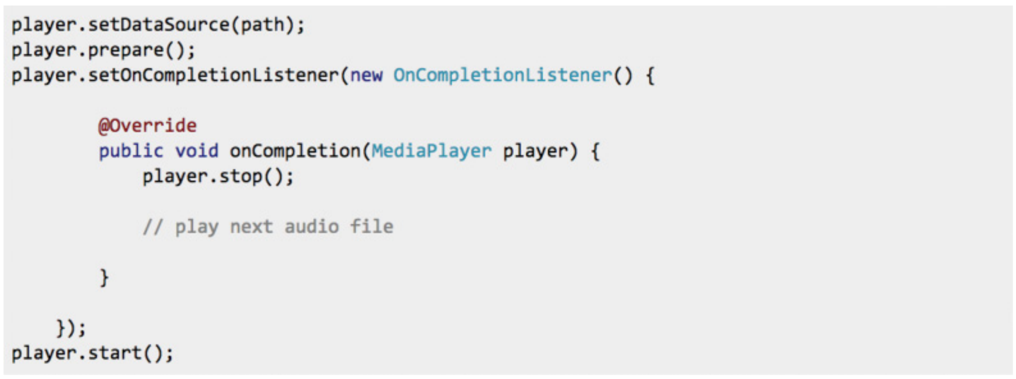
We find that the single app baseline predicts API call changes with an average precision and recall of 36% and 23%, respectively. The best source history approach is the similar apps, RA 1.3, with a precision and recall of 81% and 28%, respectively. For the Stack Overflow–based approaches, RA 1.1, the full set of co-occurring API calls in a post, has the highest precision and recall: 66% and 13%, respectively. We combine the community of apps, RA 1.2, and Stack Overflow posts, RA 1.1 and achieve an average precision and recall of 77% and 21%, respectively. We cannot use similar apps, RA 1.3, in this combined approach, as not all categories have sufficient data to make predictions and some apps would be excluded.

The API call rules that we generate are general and can be used in a wide range of applications that have been researched over the past 17 years. A few examples include identifying common API usage patterns from client applications [Michail 1999; Lamba et al. 2015], determining deviations in these usage patterns to find bugs [Livshits and Zimmermann 2005], mining programming rules and specifications [Liu et al. 2006; Kagdi et al. 2007], suggesting code completions [Bruch et al. 2009], generating usage examples and documenting APIs [Buse and Weimer 2012], identifying the degree of redundancy in software systems [Gabel and Su 2010; Ray et al. 2015; Nguyen et al. 2013a], migrating a system to a new API [Dig et al. 2006; Uddin et al. 2011] or new language [Nguyen et al. 2014], identifying how quickly client apps migrate to new APIs [McDonnell et al. 2013; Lamba et al. 2015], and understanding how API changes affect the success of apps [Linares-Vasquez et al. 2013; Guerrouj et al. 2015].



```
+    public void play(Context context) {
+        mediaPlayer = MediaPlayer.create(context, R.raw.one_small_step);
+        mediaPlayer.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {
+
+            public void onCompletion(MediaPlayer mp) {
+                mp.stop();
+            }
+        });
+        mediaPlayer.start();
+    }
```

Fig. 1. Example of API calls that change together in a commit that "added audio playback": https://github.com/jameskbride/hello-moon/commit/7b24e599586cbbc9c32710defe6d60b5deb0efef.



```
player.setDataSource(path);
player.prepare();
player.setOnCompletionListener(new OnCompletionListener() {

        @Override
        public void onCompletion(MediaPlayer player) {
            player.stop();

            // play next audio file

        }

    });
player.start();
```

Fig. 2. Example of API calls discussed together in the Stack Overflow answer to the question, "How to play media files automatically one after another?" http://stackoverflow.com/a/11291751/1055441.

The remainder of this article is structured as follows. In Section 2, we discuss our research approaches in detail and provide motivating examples of related API calls in the app version history and in Stack Overflow posts. In Section 3, we describe our methodology and data. In Section 4, we evaluate how well each of our approaches predict the changes that developers make to their apps. In Section 5, we discuss threats to validity. In Section 6, we position this work in the literature. We conclude the article in Section 7.

## 2. MOTIVATING EXAMPLES AND RESEARCH APPROACHES

Zimmermann et al. [2005] observed that related functions could be mined from the version history. We create association mining rules from the API calls that co-occur in source code changes to Android apps. These API call rules describe the changes that developers commonly make to their apps. For example, in Figure 1, we can see that `MediaPlayer.start()` and `MediaPlayer.stop()` were added in the same commit. If this pattern occurs across a large number of apps, we would create a rule that relates these two elements.

We observe on Stack Overflow posts that programmers who discussed an API call were also interested in API calls contained in the same post. We create association mining rules from API calls that co-occur in Stack Overflow discussion posts related to Android. For example, in Figure 2, we see that `MediaPlayer.start()` is discussed in the same post as `MediaPlayer.setOnCompletionListener()`. If this pattern regularly repeats in Stack Overflow posts, we would suggest that developers use `setOnCompletionListener()` when we see that they have started a `MediaPlayer`. Although researchers have used a developer's context to suggest code snippets and posts that may be useful, we are the first to use co-occurring API calls in informal documentation to suggest changes that developers should make to their apps.

We evaluate the quality of our rules by creating a test benchmark containing the co-occurring API calls in the most recent 20% of changes in our corpus of Android apps. We then use rules from previous source code change (RA 1) and Stack Overflow posts (RA 2) to predict the changes developers actually made to their apps in the benchmark. In total, we test rules mined from seven different combinations and subsets of our data.

*RA 1: Version History.* Using the version history of apps, we create a before and after version of each change. We extract the API calls that are on lines that are added or removed (a modification will have an add and remove line). We eliminate API calls that are unchanged on a line. We create association rules based on the pairs of API calls that we extract. We test the following three approaches.

*RA 1.1: Individual app baseline*. Like Zimmerman et al. [2005], we use the history of changes to the system to create rules for API calls that change together. This technique is our baseline, as all systems should have access to their own history. As Livshits and Zimmermann [2005] found, we expect to have limited predictive power because repeated calls to the same API methods should occur infrequently throughout the change history of a single app. We attain an average precision and recall of 36% and 23%, respectively.

*RA 1.2: Community of apps*. The Android API is used in a similar manner across multiple apps. We combine the rules used in individual apps to create rules from a community of applications. In this way, we are able to make suggestions to app developers on how to use API calls that they may have never used before. This research approach is most similar to those of Nguyen et al. [2013a], who predicted changes to the AST structure in a large set of Java applications. Our predictions should also contain API migration and could be used to guide migrations to new versions of an API [Uddin et al. 2011; Duala-Ekoko and Robillard 2011]. We attain an average precision and recall of 75% and 22%, respectively.

*RA 1.3: Similar apps*. The Android API allows for a wide variety of applications that serve many different purposes, from business to games. Instead of clustering rules from the entire community of apps, we cluster rules from apps in similar categories. Since similar apps will make similar calls, we expect this approach to improve the recall over the community of apps approach. Although others have used similar apps to find security violations [Gorla et al. 2014b], we are the first to use similar apps to predict future app changes. The average precision and recall was 81% and 28%, respectively.

*RA 2: Informal Documentation on StackOverflow.* Discussion on Stack Overflow describes how to combine API calls to solve problems that developers commonly face. We generate rules from the API calls present in positively voted answer posts. These rules are then used to predict the changes to individual Android apps. We have three novel subapproaches.

*RA 2.1: Stack Overflow posts*. We cluster the API calls that are contained in a single post. To extract API calls from Stack Overflow posts, we use a tool of Rigby and Robillard [2013] that can accurately identify qualified API method calls (e.g., Intent.addCategory) from natural freeform text and code snippets that do not necessarily compile. We group API calls in a post and generate association rules for API calls. We attain an average precision and recall of 66% and 13%, respectively.

*RA 2.2: Stack Overflow code snippets*. Code snippets most often demonstrate the usage of an API [Ying and Robillard 2013, 2014]. There has been much work on extracting code snippets to enhance traditional API documentation with up-to-date source code examples [Subramanian et al. 2014]. Instead of helping in the search for code examples, we use association rules from code examples to guide developers in the changes that app developers make to their apps. We attain an average precision and recall of 63% and 14%, respectively.

*RA 2.3: API calls in freeform text*. Rigby and Robillard [2013] found that code snippets contain setup code that is repeated across many posts. They found that code contained in freeform text tended to have a higher salience to the problem at hand. In contrast, code snippets tend to contain setup code that is generally necessary for development but less important for a specific problem. For this approach, we create rules only from API calls that are surrounded by natural language. For example, in Figure 2, we can see that a code element surrounded by freeform text (e.g., "You can use `MediaPlayer.onCompletionListener()` to listen to the event when a track ends . . .") is more important than contextual elements like `MediaPlayer.stop()`, which are generally useful. We attain an average precision and recall of 62% and 17%, respectively.

*RA 3: Combining the best approaches*. After evaluating the predictive power of each individual approach, we combine the top rules from the version history of a community of applications with the best Stack Overflow rules. We are unable to use the similar apps approach in the combined model because some categories do not have enough transactions to make predictions. Our goal is to determine how complementary the sets of rules are. We combine the rules from the community of apps and all Stack Overflow posts. The combined model has a precision and recall of 77% and 21%, respectively.

## 3. METHOD AND DATA

Our methodology involves the following. First, we extract the API calls from changes to apps and positively voted Stack Overflow posts. For changes, we generate a before and after state for each change and identify code elements that were added and removed. Second, we create association mining rules from the frequency of co-occurrence of API calls in changes and posts. Third, we set aside the most recent 20% of changes from our app corpus as a test benchmark. Fourth, we divide the dataset according to our research approaches and test how well each approach predicts the changes in our benchmark.

### 3.1. Stage 1: Data Preprocessing

Before extracting API call rules in app development history and Stack Overflow discussion, we need to ensure that the data is appropriate and that we have a reasonable history to mine. In the following, we present the data collection and data preprocessing steps that precede the identification of related API calls.

The first step involved downloading the change history from Git repositories. The Git repository of each application was downloaded using a Web crawler that we designed to parse the F-Droid Web pages and extract information about the Git repositories from the F-Droid catalogue.[1] The parser provides information about the Git repositories of each Android application and additional information, such as a link to the app on the Android Play Store.

Table I summarizes the main characteristics of the studied applications, including the app category (App Domains), the number of the randomly chosen applications from each category (#Apps), the number of commits (#Versions), the number of source code files that changed (#Changed Files), and the number of API call changes (#API Call Changes) per each category. We chose a random sample of 230 apps, excluding those that had few changes.[2] Unlike previous works that sampled the most recent version from thousands of apps, we had to analyze each changed code element from 12k versions and 152k changed API elements, which limited the number of apps that we could study.

In the second step, we mined the version history of each app to extract changes in API calls. We parsed each Git repository and extracted the following: changed source code files, the type of changes made in each commit (e.g., addition or removal of an

[1]https://f-droid.org/.
[2]The version history of the apps are available at http://users.encs.concordia.ca/∼pcr/apps_prediction.

Table I. Characteristics of the Studied Apps

| App Domains | App Characteristics | | | |
|---|---|---|---|---|
| | #Apps | #Versions | #Changed Files | #API Call Changes |
| Arcade & Card | 3 | 49 | 37 | 356 |
| Books & Reference | 7 | 498 | 495 | 4,048 |
| Business | 2 | 79 | 40 | 694 |
| Communication | 9 | 1,623 | 1,151 | 19,050 |
| Education | 7 | 426 | 203 | 6,060 |
| Entertainment | 8 | 705 | 457 | 6,110 |
| Finance | 5 | 640 | 658 | 9,558 |
| Health & Fitness | 3 | 73 | 35 | 736 |
| Libraries & Demo | 2 | 22 | 24 | 959 |
| Lifestyle | 3 | 61 | 48 | 421 |
| Media & Video | 6 | 790 | 694 | 7,806 |
| Music & Audio | 4 | 404 | 295 | 3,629 |
| News & Magazines | 2 | 166 | 100 | 2,188 |
| Personalization | 12 | 776 | 440 | 10,619 |
| Photography | 2 | 15 | 22 | 188 |
| Productivity | 14 | 419 | 502 | 6,024 |
| Puzzles | 6 | 95 | 84 | 624 |
| Social | 5 | 426 | 286 | 5,180 |
| Tools | 120 | 4,144 | 3,844 | 56,930 |
| Transportation | 6 | 109 | 107 | 984 |
| Travel & Local | 3 | 59 | 86 | 848 |
| Total | 230 | 12,172 | 10,180 | 152,624 |

internal or external API element), the developer who committed the change, and the time of the change. The data was stored in a PostgreSQL database to facilitate further linking and processing.

### 3.2. Stage 2: Extracting API Calls from Development History

The process of identifying fully qualified API calls is more difficult than that of identifying code elements internal to a system. To identify changing internal code elements, as Zimmerman et al. [2005] did, one simply looks for changes inside a class to, for example, the body of a method. The fully qualified name is apparent. However, with API method calls, one must resolve the type bindings to an external library. One approach is to build every version of every app. To do this, one must build up a large repository of outdated libraries and other components. Building these applications is difficult; for example, Rahman et al. [2014] talk about a "hard-won" dataset on which they built a total of 34 versions across five applications.

Since we must resolve the API calls in 12k versions of 230 apps, we use partial programming analysis (PPA) developed by Dagenais and Hendren [2008]. PPA creates an intermediate representation of the source code and returns the fully qualified names of each element. In a partial program, the complete type information may not be available, so a set of heuristics are used to extract fully qualified names, such as naming conventions and multiple passes of the partial code to resolve polymorphic methods. In some cases, ambiguity will remain, such as when the class that declares a method name is known but the package name is unknown. In experiments performed by the authors, the technique attained a precision of 91% [Dagenais and Hendren 2008].

For our purposes, PPA is not fast enough to process each file for each version of each app, so we created a pipeline that allowed us to run PPA only on changed files and to extract the fully qualified name of API calls that had changed. We used the following steps:

—First, using *git-log*, we identified the files that had changed and the lines that had changed.
—Second, for each change, we generated the state of the system before and after each change using *git-checkout*.
—Third, we ran PPA on the *before* and *after* state of each file to identify all code elements.
—Fourth, using the line numbers that had changed, we were able to identify all of the removed code elements in the before state and all of the added ones in the after state.
  If a code element occurred in both before and after states, and it was on a changed line but remained unchanged itself, it was excluded from further analysis.
—Fifth, the fully qualified name was stored in the database, indexed to its change commit. Although we have information about classes, APIs are used for their behavior and a change in a fully qualified method call will naturally cover the API classes as well.
—We calculated the co-occurrence frequency of API calls at the commit level. From these frequencies, we are able to create association mining rules relating API calls.

*3.2.1. API Calls by Application Category.* Using all changes from the version history of all apps will likely create a set of rules that are too general. As a result, we also create association mining rules for each application category. Recently, researchers have mined apps by their category, such as Business versus Education. The key idea was to associate categories and API usage to detect security anomalies—for instance, applications whose behavior would be unexpected given their categories [Gorla et al. 2014b]. Inspired by this study, we cluster applications by their category because we expect apps in the same category to use more similar API calls than those in different categories.

We categorized the analyzed Android apps based on the categories used in the Google Play Store.[3] On F-Droid and the Google Play Store, “appid” is the common identifier used. Using this identifier and Marketplace API,[4] we were able to access the information about the considered Android apps’ category. We identified 22 different categories corresponding to the analyzed Android apps. We grouped the applications together based on these categories. Each cluster consists of a different number of applications, as they were randomly chosen from F-Droid (see Table I). We group historical changes by community of applications belonging to each category, then we generate recommendations concerning co-occurring API calls using the development history of each cluster of similar applications, and finally we compute precision and recall for our recommendations for each individual app in our benchmark.

### 3.3. Stage 3: Identifying and Grouping API Calls on Stack Overflow

A great deal of knowledge about the behavior of an API is contained in the informal discussions of developers as they help each other solve problems. We extract qualified API calls (e.g., Class.method()) from Stack Overflow. Stack Overflow is a question and answer forum for professional developers.[5] Developers ask and answer questions, as well as vote on the quality of a post. Each post is related to a specific topic that involves a set of related API calls. We group the API calls found in Stack Overflow posts to create rules to predict the API calls that developers use in their apps. These co-occurring API calls provide rules that solve specific, recurring problems that app

[3]https://play.google.com/store?hl=en.
[4]https://code.google.com/p/android-market-api/.
[5]http://stackoverflow.com/tour.

developers face and complement the co-evolving API call groupings found in the version history. For example, consider the code elements in Figure 2 that show an answer post to the question, "How to play media files automatically one after another?" We see that {`MediaPlayer.setDataSource`, `MediaPlayer.prepare`, `MediaPlayer.setOnCompletionListener`, `MediaPlayer.OnCompletionListener`, `MediaPlayer.stop`, `MediaPlayer.start`} co-occur. As we discuss in the next step, if they co-occur in a large number of posts, they will become rules.

Extracting API calls from software artifacts, such as documentation and requirements, has received significant research attention. For example, information retrieval techniques, such as vector space models and latent semantic index, have been tried but have yielded low precision and recall (less than 65%) [Bacchelli et al. 2010]. In this investigation, we extract API method calls from each Android tagged Stack Overflow post. We use the automatic code element extractor (ACE) of Rigby and Robillard [2013]. ACE can extract code elements from documents that contain freeform text as well as code snippets that may not be compilable. It can process millions of post with high precision and recall (above 0.90) [Rigby and Robillard 2013]. ACE uses an island parser to identify code elements in documents. Unlike prior works, this approach does not depend on an index of valid elements parsed from the source code of a particular system [Antoniol et al. 2002; Marcus et al. 2005; Dagenais and Robillard 2012]. Instead, it identifies code elements in Java constructs and creates an index of valid elements based on the elements contained in the collection of documents. More recent works, which do not need an index of valid terms, can only parse code snippets and miss code elements that are in freeform text [Subramanian et al. 2014].

In this work, the API calls in a Stack Overflow post are considered to "co-occur." Since some posts are of low quality, we only consider posts that have received more positive votes than negative votes—that is, they are positively voted. Further, we exclude question posts because questioners do not know where to focus and so provide as much information as possible in the hope that someone will spot their problem. Question posts would introduce noise in our data and impact the accuracy of our predictions. We also eliminate any API calls that are part of stack traces, as most of the elements in a dump are not relevant to the post.

We processed the Stack Overflow posts tagged with "android" between August 2008 and October 2014. We filtered the posts based on our three Stack Overflow research approaches. Since we are creating rules between API calls, we only consider posts with at least two API calls. The first approach leverages the entire content of each positively voted Stack Overflow answer: 113k answer posts. The second approach exploits API calls in code snippets only: 84k posts. The third approach focuses on API calls that are mentioned in natural language text exclusively: 29k posts.

## 3.4. Step 4: Association Rule Mining

We use association rule mining to find co-occurring API calls that we can suggest to app developers. To generate rules, we need to understand how often items (API calls) co-occur in transactions (version control commits or Stack Overflow posts). Ultimately, we will generate rules of the form $X \Rightarrow Y$, where $X$ is called the *antecedent* (if) and $Y$ the *consequent* (then).

For example, on Stack Overflow answer posts, we find that the API calls `MediaPlayer.start()` and `MediaPlayer.stop()` have been discussed together in 724 posts, whereas `MediaPlayer.start()` and `MediaPlayer.setOnCompletionListener` have been discussed together in 463 posts. One such post is shown in Figure 2. Both associations are strong, with the association between start and stop being stronger. We would create two rules, with the following being an example of the latter association:

$$MediaPlayer.start \Rightarrow MediaPlayer.setOnCompletionListener.$$

This rule implies that when a developer changes the API call `MediaPlayer.start()`, he or she should be aware of the call `MediaPlayer.setOnCompletionListener()`, as they are often discussed together on Stack Overflow.

The rules that we generate are probabilistic in nature, so we quantify co-occurrence using confidence and support count. *Confidence* is the number of transactions that $X$ and $Y$ co-occur in divided by the total number of transactions in which $X$ occurs.

$$confidence(X \Rightarrow Y) = \frac{frequency(X \cup Y)}{frequency(X)}$$

We follow Zimmerman et al. [2005] in our use of support count instead of support. The support count is simply the number of transactions in which both $X$ and $Y$ occur.

$$support\_count(X \Rightarrow Y) = frequency(X \cup Y)$$

*3.4.1. Generating Rules Using Frequent Pairs.* The apriori algorithm computes all rules beforehand, and rules with support and confidence above a threshold are kept. Since computing all possible rules can be time consuming, up to 2 to 3 days in our experiments, following Zimmerman et al. [2005], we optimize the computation by making the following modifications. We only consider single antecedents. In our example, we can see that we have a single antecedent—that is, there is only one API call on the left-hand side. Association rules with more than one item in their antecedents, such as $api_1$, $api_2 \Rightarrow api_3$, are not considered.

We also compute rules with a single item in their consequent. For a given item (e.g., *api1*), the rules have the form $api_1 \Rightarrow api_2$.

### 3.5. Evaluation Setup

We divide our data into training and test data.[6] The training data is used to generate rules relating API calls. These rules are then used to predict the API calls that co-evolve in individual apps in the test dataset.

We use the same test dataset to evaluate the quality of the rules generated from both the version history and the Stack Overflow datasets. The test set consists of the most recent 20% of commits in the version history of the apps (2,435 transactions and 30,223 items). The version control training set consists of the remaining, older 80% of the data. The Stack Overflow training set consists of all Stack Overflow posts.

We evaluate the quality of our predictions as follows. We divide each transaction from our test dataset into two parts: (1) a query, $q$, and (2) an expected outcome, $E_q$. If a transaction consists of $n$ items, then in the first pass we take the first item $i_1$ as the query and the remaining items $i_2, i_3, \ldots, i_n$ as the expected outcome. In the second pass, the query will be $i_2$ and $i_1, i_3, \ldots, i_n$ will be expected outcome. We perform $n$ passes.

Having generated all possible queries and expected outcomes from our test dataset, we take each item in our test set as an antecedent in our training set. Using this antecedent, we take the top 10 consequents based on the confidence of the derived rules, $C_q$. Our calculation of precision and recall is dependent on how many of these consequents are in the expected outcome, $E_q$, of our test set. The precision, $P_q$, and

[6]We also conduct 10-fold cross validation. See Section 5.

recall, $R_q$, for a given query will be

$$\begin{aligned} P_q &= |C_q \cap E_q|/|C_q| \\ R_q &= |C_q \cap E_q|/|E_q|. \end{aligned}$$

To measure the performance across all queries in a transaction, we calculate the average precision and recall. To compare the predictive power of each research approach, such as rules generated from version history versus Stack Overflow, we conduct pairwise comparisons of the precision and recall using the Wilcoxon rank sum test or Mann-Whitney test. We compare the baseline approach and current best approach instead of generating comparisons between every possible combination. In cases of more than two comparisons, we use the Kruskal-Wallis test.

### 3.6. Performance: Time and Resources

We summarize our approach and discuss the necessary resources. All data processing was performed on a standard desktop machine. Our approach can be broken into three stages: a per-app stage, a Stack Overflow stage, and a rule generation stage. The first stage is only run once per app and has the following steps:

(1) *Clone the git repository*. The performance of this step is dependent on Git and the number and size of repositories.
(2) *Parse each change*. We walk the Git dag creating a before and after state. This must be done once for each commit/change. The processing involved in this stage is expensive, so we randomly sampled 230 apps and processed 12k commits. Using a more involved approach, Rahman et al. [2014] were able to process only 34 versions across five applications. When new changes are made to an app, only the new changes need to be parsed.
(3) *Run PPA on each change*. PPA creates a partial AST. We then determine which code elements are on changed lines. This is the most expensive stage of our analysis, and it is dependent on the size of the change. The largest changes ran for up to 1 minute. We extracted 152k API calls.
(4) *Create transaction based on commits*. Grouping the API calls by commit takes a trivial amount of time.

The second stage involves processing Stack Overflow posts. We use Rigby and Robillard's ACE tool, which is substantially faster than previous work [Rigby and Robillard 2013]. However, in running the tool on 500k Android posts on Stack Overflow, it took 1.5 weeks to finish. We are currently making simple adaptations to the tool to allow it to progressively ingest posts.

The third stage involves generating the rules between API calls. Every time a new app is added, a new change is made to an existing app, or a new post is made on Stack Overflow, we must regenerate the rules. It took us a maximum of 6 hours to generate all of the rules on our complete Stack Overflow and app history datasets. The new rules could be generated nightly or when a large set of changes had occurred. Since we follow Zimmermann et al. [2005] and generate rules in a pairwise manner, we would be able to only update rules that contain changed API calls, reducing the set of rules that need to be generated.

## 4. EMPIRICAL EVALUATION

We use rules generated from the version control history of apps and Android answer posts to predict the changes in API calls that individual developers make to their apps. The test set remains the same for all of our research approaches and is the most recent

20% of commits made to the apps under study. The test set contains 2,435 transactions (commits) and 30,223 API calls (items).

### 4.1. RA 1: Version History

In this section, we use the changes that app developers have made in the past to predict the changes that will be made in the future. We group API calls that change together using association mining rules. We have three approaches. First, we create a baseline, making predictions using single apps. Second, we combine the rules generated across the entire community of apps. Third, we combine rules that come from apps in the same category. The test set remains constant across all approaches. The training set for the version history approaches consists of 9,266 transactions and 51,908 items.

*4.1.1. RA 1.1: Individual App Baseline.* Our baseline mirrors the work of Zimmerman et al. [2005] by using past source code changes to predict the changes that an app developer will make to the same app in the future. We had 27 transactions per app on average for the training set and 8 transactions on average for the test set. Each transaction consists of an average of six items.

Since we have the development history for 230 different individual apps, we had to perform several experiments. To facilitate comparison across all applications, we decided on a common value for all apps as in previous works [Zimmermann et al. 2005]. We choose thresholds that are not too low and not to high to ensure a trade-off between the number of rules and their relevance. The support count threshold is 7 and a confidence threshold is 50%.

Figure 3 shows the distribution of precision and recall for our version history approaches. Our single app version history baseline is able to predict API call changes with an average precision of 36% and recall of 23%. The distribution is skewed toward lower values. Although relatively low, these results are consistent with Zimmerman et al. [2004], who reported an average precision and recall of 29% and 44%, respectively. With the small size of apps, we might expect substantially lower results than Zimmerman et al., who used large systems like Eclipse and JBoss that have a long history. We suspect that one reason we achieved better precision (7 points higher) and lower recall (21 points lower) is because we are predicting API calls and not changes to internal methods. The possible set of rules is smaller with API calls, so our predictions may be accurate despite a short version history.

*4.1.2. RA 1.2: Community of Apps.* Our baseline produced a reasonable precision and recall despite a short version history. Since API calls are used in similar patterns across multiple apps, we combine the rules generated from the version history of all apps to predict the API calls that will be changed together by individual app developers.

As in our first approach, we experimentally determined the minimum support and confidence thresholds suitable for our training set by means of several experiments startingfrom low support and confidence thresholds up to high ones, then we chose the final parameters that enable us to find a compromise between the number of generated rules and their pertinence. The configuration chosen consists of a support count of 15 and a confidence of 75%.

The distributions in Figure 3 show that we can predict changes in individual apps with an average precision of 75% and an average recall of 22%. Compared to our baseline, there is no statistically significant difference in recall, whereas precision increases by 39 points ($p < 0.001$).

Given the task of suggesting possibly relevant API calls to developers, we suggest a related call that the developer actually used 75% of the time. The high precision clearly illustrates that developers use API calls in very regular and consistent patterns.

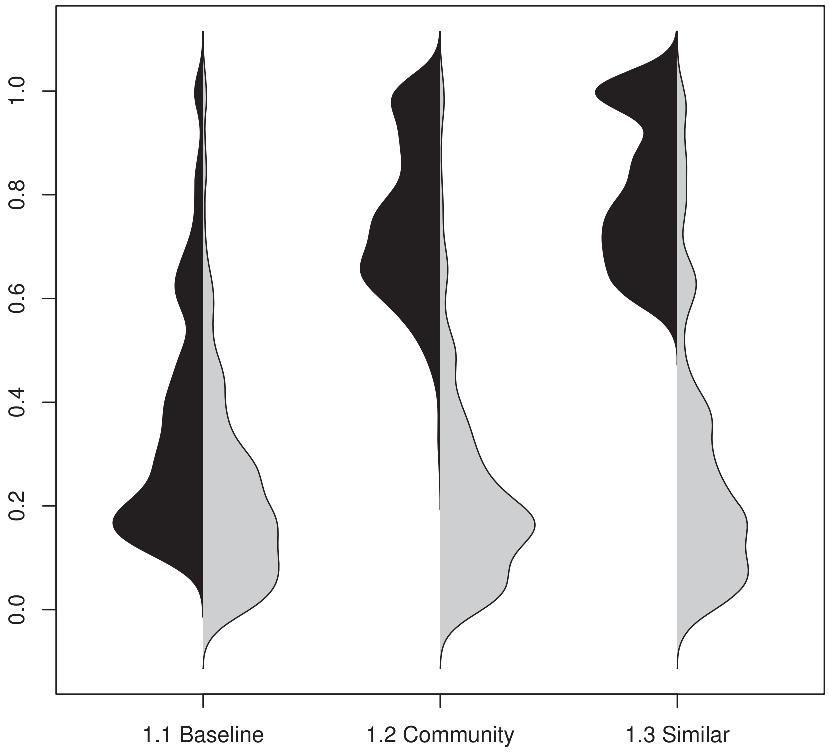


Fig. 3. Distribution of precision (left) and recall (right) for the version history research approaches: RA 1.1 is the baseline that predicts future changes based on the history of a single app, RA 1.2 uses the rules from the entire community of apps to predict changes to individual apps, and RA 1.3 uses rules from apps in the same category to predict changes to individual apps.

The low recall indicates that there are many different ways to combine API calls, and although we accurately suggest related API calls, we miss many of the possible combinations. Since we are only suggesting the top 10 related API calls, we often miss calls that developers actually end up using. We suspect that the main problem relates to the diversity of apps in our sample. For example, a weather app might use the GPS location in a very different way than a traffic app.

*4.1.3. RA 1.3: Similar Apps.* Our goal is to improve recall while keeping precision high. We cluster apps by categories to get rid of unrelated API changes in our rules. In Table II, we show the 11 app clusters, from business to travel. We eliminated categories, such as Lifestyle, that did not have 85 or more transactions in the test dataset. Table II also shows the number of source code files changed, the number of transactions, and the number of items per each category of app.

Since we have the version history for 11 different categories, we had to perform several experiments with different minimum support and confidence thresholds for each category. We selected a minimum support count and confidence thresholds of 15 and 60%, respectively.

Using the rules generated from apps in the same category to predict co-changing API calls, we can predict changes in individual apps with an average precision of 81% and an average recall of 28% (see the distributions in Figure 3). Compared to the rules generated from the entire community of apps, we see an increase in average precision

Table II. Development History of Analyzed Projects

| App Domains | Training Data | | Test Data | |
|---|---|---|---|---|
| Category | #Txns | #Items | #Txns | #Items |
| Books & Reference | 295 | 1,047 | 101 | 263 |
| Communication | 968 | 5,430 | 327 | 1,289 |
| Education | 249 | 956 | 88 | 291 |
| Entertainment | 445 | 1,088 | 145 | 794 |
| Finance | 422 | 2,644 | 144 | 946 |
| Media & Video | 486 | 1874 | 167 | 761 |
| Music & Audio | 250 | 960 | 88 | 419 |
| Personalization | 457 | 2,732 | 159 | 657 |
| Social | 250 | 1,445 | 88 | 328 |
| Tools | 2,556 | 16,376 | 929 | 5,572 |
| Transportation | 103 | 408 | 41 | 177 |

Txns, transactions.

of 3 points, but we improve our average recall by 6 points. The difference for precision is not statistically significant with $p = 0.07$, whereas the increase in recall is $p = 0.04$.

Our results confirm that similar apps use similar API calls. However, using similar categories of apps reduces the sample from which we can draw and in some categories makes prediction impractical. We suspect that the modest increase in recall is the result of more specific sample of related apps, which reduces the overall number of possible API call combinations. Since we are unable to make predictions for all apps, we continue to use the community of apps as our comparison point for Stack Overflow.

### 4.2. RA 2: Informal API Documentation on Stack Overflow

Informal API documentation contains rich information about API calls [Ying and Robillard 2014; Subramanian et al. 2014; Rigby and Robillard 2013]. We use Stack Overflow to predict the changes that developers will make to their apps. The test set remains the same and contains the 2,435 transactions and 30,223 items from version history. The training set is the API calls that are contained in the same Stack Overflow post.

*4.2.1. RA 2.1: Stack Overflow Posts.* We first consider the entire content of positively voted answers. Our training set consist of all API calls mentioned in both the code snippets and freeform text of Stack Overflow posts. We have a total of 113,303 transactions (posts) and 406,622 items (API calls) from our training set. We have experimentally set the support count and confidence thresholds prior to 10% and 70%, respectively.

We find that we can predict changes in individual apps with an average precision of 66% and an average recall of 13% (see the distributions in Figure 4). Compared to our baseline, we increase our precision by 31 points but decrease our recall by 10 points. Both results are statistically significant with $p \ll 0.001$. Compared to the version history from a community of apps, RA 1.2, we see a decrease in both precision and recall of 9 points ($p < 0.01$). Comparing all three approaches using a Kruskal-Wallis test, the differences are statistically significant with $p \ll 0.001$.

In Figure 4, we see that the recall distribution is skewed toward low values. We suggest two possible factors. First, we only include the top 10 results in our predication. Second, the Stack Overflow dataset has eight times as many API calls as the version history dataset. As a result, we suspect that the large dataset has too many possible suggestions for each API call, which limits the recall of the Stack Overflow approach. In contrast, the suggestions made from the Stack Overflow data are precise, with the distribution heavily skewed toward high values of precision.

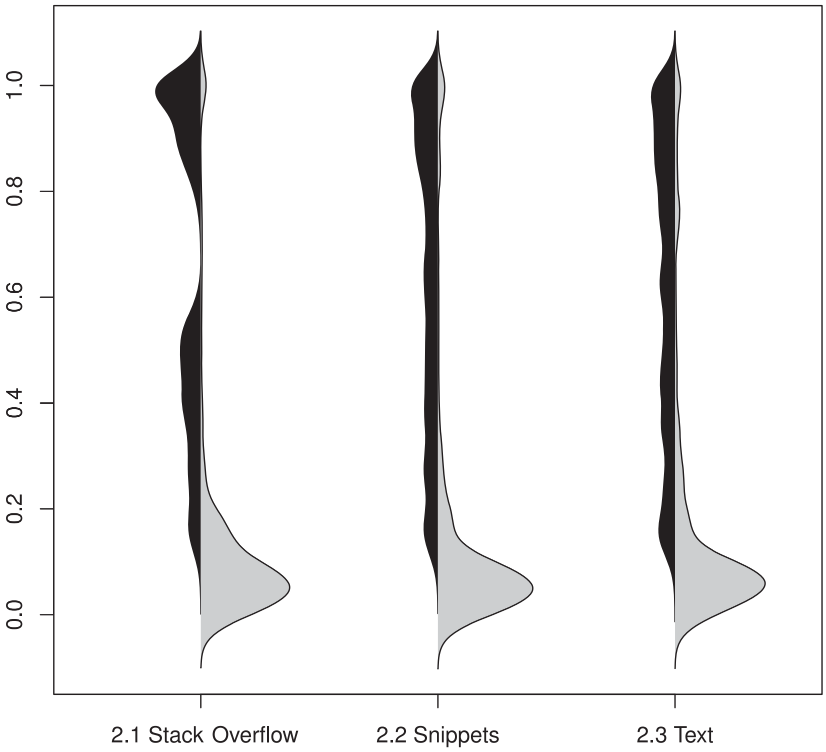


Fig. 4. Distribution of precision (left) and recall (right) for the Stack Overflow research approaches: RA 2.1 includes all API calls in a post, RA 2.2 includes only API calls in code snippets, and RA 2.3 includes only API calls surrounded by freeform text. In each case, we predict the changes that will be made to individual apps.

*4.2.2. RA 2.2: Stack Overflow Code Snippets.* Code snippets help developers learn and relearn proper API usage patterns [Ying and Robillard 2014; Subramanian et al. 2014]. Researchers have shown that 65% of accepted answers on Stack Overflow contain code examples [Subramanian and Holmes 2013], whereas unanswered questions often lack code [Asaduzzaman et al. 2013]. Our goal is to understand whether code snippets contain related API calls that individual developers ultimately use in their apps.

Mining only Stack Overflow code snippets, our training set contained 84,189 transactions and 361,884 items. By limiting our set to code snippets, we reduce the number of API calls by 45k. Our test set remains the same. We experimentally set our minimum support count and a minimum confidence to 15 and 65%, respectively.

The distribution in Figure 4 show that by using code snippets, we can predict changes in individual apps with an average precision and recall of 63% and 14%, respectively. Compared to the previous approach that included API calls in code snippets and freeform text, the differences are not statistically significant ($p > 0.13$).

*4.2.3. RA 2.2: Methods in Freeform Text on Stack Overflow.* Rigby and Robillard [2013] found that the API calls present in the freeform text are more central (i.e., salient) to the subject of a post than code snippets. Code snippets contain setup code that occurs repeatedly across posts. We suspect that this setup code adds noise to predictions. We investigate whether API calls contained in freeform text on Stack Overflow provide a more accurate prediction of the API calls that developers use in their apps.

Table III. Average Precision and Recall for each Research Approach

| Research Approach | Data Source | Precision (%) | Recall (%) |
|---|---|---|---|
| RA 1.1 | Single app baseline | 36 | 23 |
| RA 1.2 | Community of apps | 75 | 22 |
| RA 1.3 | Similar apps | 81 | 28 |
| RA 2.1 | Stack Overflow posts | 66 | 13 |
| RA 2.1 | Stack Overflow code snippets | 63 | 14 |
| RA 2.1 | Stack Overflow freeform text | 62 | 17 |
| RA 3 | Combined: community and all posts | 77 | 21 |

Our training set consists of all clusters of API calls present in natural language text of Stack Overflow answers posts exclusively. It consists of a total of 29,114 transactions and 47,021 items. By limiting our set to those contained in freeform text, we reduce the number of items in our training set 8.6 times, from 406k to 47k. The test set remains the same as in the previous approaches. We experimentally choose a support count of 15 and a minimum confidence of 70%.

Identifying API calls from freeform text, we can predict changes in individual apps with an average precision of 62% and an average recall of 17% (see the distributions in Figure 4). Compared to the snippets-only approach, the change in precision is not statistically significant ($p = 0.50$), whereas there is an increase in recall of 3 points ($p = 0.02$). In contrast to the approach that contains both code snippet and text, we decrease the precision by 4 points ($p = 0.03$) but increase the recall by 4 points ($p = 0.08$). The increase in recall is only a trend (i.e., $p < 0.10$). Comparing all three approaches using a Kruskal-Wallis test, the differences are not statistically significant, with $p > 0.05$ for both precision and recall.

The slight increase in recall likely arises from a more focused set of API calls—that is, those described in freeform text. We suspect that the difference in precision results from missing contextual API calls that are present in code snippets. These contextual API calls are relevant to many tasks, and a narrow focus on freeform text misses some related elements.

## 4.3. RA 3: Combing the Best Approaches

Table III shows the precision and recall for each approach. We create a model that integrates the best models from version history and Stack Overflow posts. We combine the version history from the community of all apps, RA 1.2, with the rules generated from all API calls contained in Stack Overflow posts. We do not use the similar apps approach, as we cannot make a prediction for all apps. In Figure 5, we see that the combined approach has an average precision and recall of 77% and 21%, respectively.

We compare the combined approach with our baseline and its two component parts. Comparing all four approaches using a Kruskal-Wallis test indicates that the difference in precision and recall are statistically significant at $p \ll 0.001$. We also perform a pairwise comparisons using Wilcoxon rank sum tests. Compared to the average baseline predictions within a single app, the combined approach increases precision by 41 points ($p \ll 0.001$), whereas the difference in recall is not statistically significant. Compared to the community of apps, RA 1.2, we see an increase of 2 points in average precision. However, this increase is a non–statistically significant trend ($p = 0.07$). The change in recall is not statistically significant ($p = 0.85$). Compared to the rules generated from API calls on Stack Overflow, we see an increase in average precision and recall of 9 and 8 points, respectively ($p < 0.01$).

The combined model indicates a slight increase in precision over the community of apps model. The model is better than the basic Stack Overflow model. By adding more focused rules to the Stack Overflow model, we see increases in predictive power.

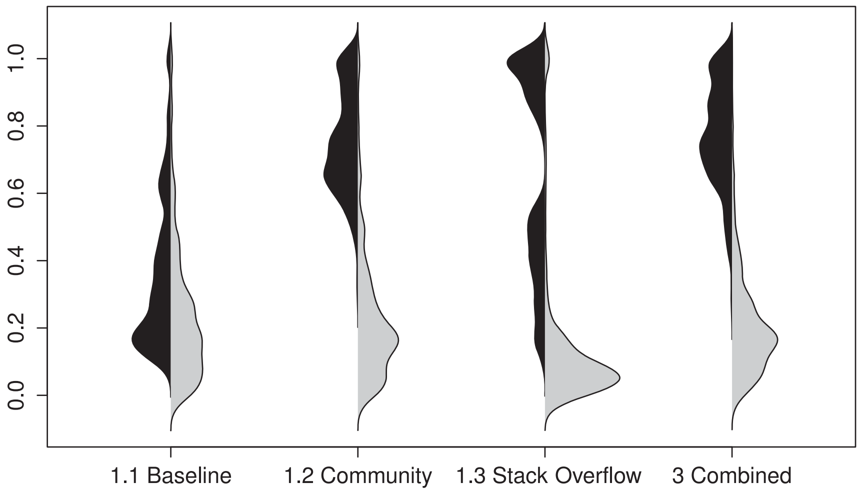


Fig. 5. Distribution of precision (left) and recall (right) for the combined approach. We combined the rules from version history of a community of apps with those generated from the API calls present in Stack Overflow posts.

However, the overwhelming number of API calls on Stack Overflow (406k calls) generates an overabundance of possible combinations. Using the context of the app developer to limit the Stack Overflow rules is a promising area of future work that may improve recall.

## 5. THREATS TO VALIDITY

In the sections where each approach was presented, we discussed the particular threats to validity for that approach. In this section, we discuss general threats to the entire study.

Although our study involves fewer apps than others studies [Vasquez et al. 2013; Gorla et al. 2014a], we have analyzed multiple version of apps instead of a single release. In total, we examined 152k commits across 230 apps. Since we have randomly sampled across a wide range of apps from 22 different categories, we believe that our results will likely generalize. It is unclear whether our results would generalize outside of Android app development, and future work is necessary.

We have examined the predictive power of our approach based not only on the use of development history but also informal API documentation. We chose Stack Overflow because it contains discussions among professional developers. However, other sources of information can be leveraged as well, including bug reports repositories, developers' emails, code reviews, and instant message discussions.

To identify API calls in app version histories, we used partial program analysis [Dagenais and Hendren 2008]. Although there are some unknown code elements, we believe that the accuracy of 91% is reasonable for our approach. Likewise, we believe that the tool of Rigby and Robillard [2013], which has an average precision and recall at or above 90% when analyzing Android Stack Overflow posts, is reasonable for our purposes.

We chose to evaluate the precision and recall based on the top 10 results returned by our approach. The top 10 results may have reduced our recall values, so we also evaluated the top 15 results. We found that compared to the top 10 results, the top 15 results decreased the precision between 4 and 6 percentage points. Except for salient

Stack Overflow API calls, all decreases are statistically significant at $p < 0.05$. The recall values increased by approximately 1 point. None of the increases in recall were statistically significant for the version history approaches. All Stack Overflow approaches had statistically significant increases in recall at $p < 0.05$. However, the increase is between 1.1 and 1.6 percentage points, which is negligible given the task. Since the recall increases negligibly with more results, we conclude that the diversity of each change is great; however, the good precision results indicate that many of the changed elements are similar. In effect, developers use many of the same API calls but also use a smaller specific set that likely makes their app different from that of competitors. We can predict the ones that they use regularly but have difficulty predicting the complete set. Future techniques based on natural language processing of commit logs may help in specifying the API calls that differentiate the app from its competitors.

To understand how useful our patterns are as general rules for API usage, we also conducted 10-fold cross validations and compared them to our time-ordered training and test set evaluations. For each research approach, there was a reduction in precision and recall. However, the difference in resulting precision and recall was not statistically significant according to a Wilcoxon rank sum test, except in the precision of the similar apps approach (average precision –3.7 points, $p < .003$) and recall of the salient Stack Overflow API calls approach (recall –2.3 points, $p < 0.36$). Although rarely statistically significant, part of this negative difference results from suggesting API calls from the future folds that did not exist in the past folds. In the body of the article, we select an older training set and a newer test set to avoid suggesting API calls that do not yet exist.

In this work, we may suggest legacy or deprecated API calls. An interesting area of future work could be to investigate weighting and other approaches to ensure that suggestions accurately reflect the current state of app development. If a recency weight is added, precision and recall should increase over the values that we report, as fewer deprecated API calls will be suggested.

## 6. RELATED WORK

Source code has long been used to identify related changes and to provide a better understanding of systems and libraries. For example, textual similarity of program code [Atkins 1998], commit messages [Chen et al. 2001], and API usage patterns [Michail 1999] have been exploited to guide developers during their engineering activities. Building on this work, Zimmerman et al. [2004, 2005] used association rule mining on CVS data to recommend source code that is potentially relevant to a given change task. Like Zimmerman et al. [2005], we apply data mining techniques and use association rule mining. In our approach, we predict changes that developers make to API method calls instead of predicting change associations between internal files or calls.

Several works have examined the redundancy of code used in a community of applications [Gabel and Su 2010], as well as in the changes made to those applications. For example, Ray et al. [2015] examine these changes at the line level, looking for changed lines that are clones, and Lamba et al. [2015] determine the degree of redundancy in Android apps. In contrast, we extract the API calls to suggest future changes. Nguyen et al. [2013a] also use version history to make predictions but make them at the AST level. AST-level predictions include other constructs, such as “for” loops, making it difficult to achieve a high degree of precision. Furthermore, we leverage informal API documentation (i.e., Stack Overflow) to find related API calls. We also propose a predictive model using both development history and Stack Overflow posts to predict changes in APIs calls.

Advanced autocompletion techniques have leveraged the history of applications and the repetitive nature of programming to suggest code elements to developers. Robbes and Lanza [2008] filter the suggestions made by code completion algorithms based on, for example, where the developer had been working in the past and the changes that he or she had made. Bruch et al. [2009] suggest appropriate method calls for a variable based on an existing code base that makes similar calls to a library. Duala-Ekoko and Robillard [2011] use structural relationships between API elements, such as the method responsible for creating a class, to suggest related elements to developers. Works by Nguyen et al. [2013b] use statistical language models to accurately autocomplete code. Since these models are built on a corpora of applications, they are able to make suggestions that autocomplete multiple lines of code. The language models suggest simple sequences, which can lead to incorrect completions. Recent work by Nguyen and Nguyen [2015] uses graphs to ensure that the suggestions are syntactically valid. These works clearly show how related method calls can be suggested to the developer as autocompletions. A simple tool would be able to use our rules to suggest changes not only from a sample of related applications but also from Stack Overflow.

Focusing on API evolution, many researchers [Dig et al. 2006; Uddin et al. 2011] have identified the changes that must be made to client programs when the API evolves. Their technique identifies temporal API usage patterns in client applications to understand when usage of API calls change. We are interested in all related API calls, not just changes related to the evolution of an API.

Several works suggest code snippets that are related to each other or to a developer's current context [Ying and Robillard 2013; Subramanian et al. 2014]. For example, Strathcona uses the developer's context to find related code [Holmes and Murphy 2005]. MAPO clusters similar code across a set of 20 open source applications to suggest related code snippets [Zhong et al. 2009]. Ponzanelli et al. [2014] are able to suggest code snippets to a developer given the context of the code elements that are open in the Eclipse IDE. During maintenance and development tasks, they showed that the suggested code snippets helped developers to complete experimentally assigned tasks. Ponzanelli et al. [2013] have developed an Eclipse plugin namely, Seahawk, that assist programmers by suggesting StackOverflow posts relevant to the context of the developer's IDE. Instead of returning existing code snippets, Buse and Weimer [2012] automatically generate code snippets from a large corpus of applications that use an API. In a user study, the generated code snippets were ranked equal to manually created snippets 82% of the time. In contrast to these works, we suggest related API elements instead of suggesting entire posts or snippets of code. Our suggestions crosscut multiple posts instead of suggesting individual posts and compare Stack Overflow to version history.

Examining the stability of an API, Linares-Vasquez et al. [2013] studied the impact of fault- and change-prone API calls on user ratings of apps. They found that apps with lower ratings used API calls that were buggy and change prone. The authors mined the version history of the Android API and related it to single releases of apps. In contrast, we mined the version history of apps that use the API. As a result, whereas they mined many more apps, we mined many more versions of apps. Their conclusion is that app developers should avoid unstable API calls, whereas we suggest API calls that commonly occur together in a large community of apps. In previous work, we examined the binaries of Play Store apps to determine if success and churn were related [Guerrouj et al. 2015]. We found that individual apps that changed frequently had lower ratings. We decompiled binary apps at releases instead of using the source version history. This methodology was not appropriate for this study, as release information is not fine grained enough to provide association rules between co-changing API calls.

In another work, Linares-Vasquez [2014] examined whether changes to the Android API would lead to increased discussion of changed API calls on Stack Overflow. They found that developers react to changes in the Android API as measured by changes to their apps and increased Stack Overflow discussions. We found a similar result [Guerrouj et al. 2015]. Neither study used Stack Overflow posts to suggest groupings of API calls as we do in this study.

## 7. CONCLUSION

In this article, we mined source code change history and Stack Overflow posts to help developers identify relevant API method calls when making changes. Using 230 apps and 12k revisions to those apps, we have shown that our techniques are able to accurately provide change suggestions of API calls that developers actually make to their apps with a high precision and recall. Making predictions using a single apps baseline, we found a precision and recall of 36% and 23%, respectively. Using a community of apps, we attained a precision of 75% and a reasonable recall of 22%. Additionally, we combined the predictive models from development history and Stack Overflow and showed that the combined model has similar performance to our version history approach. We also provided evidence that source code changes history across multiple apps in the same category, such as Finance or Games, improves the precision and recall to 81% and 28%, respectively. However, some categories did not contain enough apps to make reasonable predictions.

In terms of informal API documentation on Stack Overflow, we showed that developer posts reflect code change activities. This finding corroborates recent works [Linares-Vásquez et al. 2014; Panichella et al. 2014; Guerrouj et al. 2015]. We demonstrated that these relationships can be used to predict API call changes to individual apps with an average precision and recall of 66% and 13%, respectively. We believe that our approach can be used to augment existing works on the prediction of changes between fine-grained source code entities, as well as syntactic and dynamic analysis-based techniques.

As far as we know, we are the first to use co-occurrences of API elements on Stack Overflow to suggest API change patterns to developers. Whereas others have mined code snippets and suggested relevant posts to a developers context [Ponzanelli et al. 2014], we grouped co-occurring API elements on Stack Overflow and used these co-occurrences to predict changes that developers are likely to make to their apps. Our rules are general and can be used to suggest code completions and help in creating documentation.

## REFERENCES

G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. 2002. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 28, 10, 970–983.

Muhammad Asaduzzaman, Ahmed Shah Mashiyat, Chanchal K. Roy, and Kevin A. Schneider. 2013. Answering questions about unanswered questions of Stack Overflow. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*. IEEE, Los Alamitos, CA, 97–100.

David L. Atkins. 1998. Version sensitive editing: Change history as a programming tool. In *System Configuration Management*. Lecture Notes in Computer Science, Vol. 1439. Springer, 146–157.

A. Bacchelli, M. Lanza, and R. Robbes. 2010. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. 375–384.

Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. 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)*. ACM, New York, NY, 213–222. DOI:http://dx.doi.org/10.1145/1595696.1595728

Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API usage examples. In *Proceedings of the 20th International Conference on Software Engineering*. 782–792.

Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. 2001. CVSSearch: Searching through source code using CVS comments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. 364.

Barthelemy Dagenais and Laurie J. Hendren. 2008. Enabling static analysis for partial Java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'08)*. ACM, New York, NY, 313–328.

Barthelemy Dagenais and Martin P. Robillard. 2012. Recovering traceability links between an API and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, Los Alamitos, CA, 47–57.

Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated detection of refactorings in evolving components. In *ECOOP 2006—Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 4067. Springer, 404–428. DOI:http://dx.doi.org/10.1007/11785477_24

Ekwa Duala-Ekoko and Martin P. Robillard. 2011. Using structure-based recommendations to facilitate discoverability in APIs. In *ECOOP 2011—Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 6813.Springer, 79–104. DOI:http://dx.doi.org/10.1007/978-3-642-22655-7_5

Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, New York, NY, 147–156.

Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014a. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 1025–1035. DOI:http://dx.doi.org/10.1145/2568225.2568276

Latifa Guerrouj, Shams Azad, and Peter C. Rigby. 2015. The influence of app churn on app success and Stack Overflow discussions. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*. IEEE, Los Alamitos, CA, 10.

Reid Holmes and Gail C. Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, NY, 117–125. DOI:http://dx.doi.org/10.1145/1062455.1062491

Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. 2007. An approach to mining call-usage patterns with syntactic context. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. ACM, New York, NY, 457–460. DOI:http://dx.doi.org/10.1145/1321631.1321708

L. Gorla, I. Tavecchia, F. Gross, and A. Zeller. 2014b. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 1025–1035.

Yash Lamba, Manisha Khattar, and Ashish Sureka. 2015. Pravaaha: Mining Android applications for discovering API call usage patterns and trends. In *Proceedings of the 8th India Software Engineering Conference*. ACM, New York, NY, 10–19.

Mario Linares-Vasquez, Gabriele Bavota, Carlos Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. 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'13)*. 477–487.

Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How do API changes trigger Stack Overflow discussions? A study on the Android SDK. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. ACM, New York, NY, 83–94. DOI:http://dx.doi.org/10.1145/2597008.2597155

Chang Liu, En Ye, and Debra J. Richardson. 2006. LtRules: An automated software library usage rule extraction tool. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York, NY, 823–826.

Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, New York, NY, 296–305. DOI:http://dx.doi.org/10.1145/1081706.1081754

A. Marcus, J. I. Maletic, and A. Sergeyev. 2005. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering* 15, 4, 811–836.

Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of API stability and adoption in the Android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM'13)*. IEEE, Los Alamitos, CA, 70–79. DOI:http://dx.doi.org/10.1109/ICSM.2013.18

Amir Michail. 1999. Data mining library reuse patterns in user-selected applications. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99)*. IEEE, Los Alamitos, CA, 24. http://dl.acm.org/citation.cfm?id=519308.786921

Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE*

*International Conference on Automated Software Engineering (ASE'14)*. ACM, New York, NY, 457–468. DOI:http://dx.doi.org/10.1145/2642937.2643010

Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering, Vol. 1* (ICSE'15). 858–868.

H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. 2013a. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13)*. 180–190. DOI:http://dx.doi.org/10.1109/ASE.2013.6693078

Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013b. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, New York, NY, 532–542. DOI:http://dx.doi.org/10.1145/2491411.2491458

Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. 2014. How developers' collaborations identified from different sources tell us about code changes. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*. 251–260. DOI:http://dx.doi.org/10.1109/ICSME.2014.47

Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack Overflow in the IDE. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*. 1295–1298.

Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. 2014. Mining Stack Overflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. ACM, New York, NY, 102–111. DOI:http://dx.doi.org/10.1145/2597073.2597077

Foyzur Rahman, Sameer Khatri, Earl T. Barr, and Premkumar Devanbu. 2014. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 424–434. DOI:http://dx.doi.org/10.1145/2568225.2568269

Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. 2015. The uniqueness of changes: Characteristics and applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR'15)*. IEEE, Los Alamitos, CA, 34–44. http://dl.acm.org/citation.cfm?id=2820518.2820526

Peter C. Rigby and Martin P. Robillard. 2013. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*. 832–841.

R. Robbes and M. Lanza. 2008. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. IEEE, Los Alamitos, CA, 317–326. DOI:http://dx.doi.org/10.1109/ASE.2008.42

M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. 2013. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5, 613–637.

Martin Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6, 703–732. http://dx.doi.org/10.1007/s10664-010-9150-8

Siddharth Subramanian and Reid Holmes. 2013. Making sense of online code snippets. In *Proceedings of the 10th International Working Conference on Mining Software Repositories*. IEEE, Los Alamitos, CA, 85–88.

Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 643–652. DOI:http://dx.doi.org/10.1145/2568225.2568313

Gias Uddin, Barthlmy Dagenais, and Martin P. Robillard. 2011. Analyzing temporal API usage patterns. In *Proceedings of the 2012 34th International Conference on Software Engineering (ICSE'11)*. IEEE, Los Alamitos, CA, 456–459.

Mario Linares Vasquez, Gabriele Bavota, Carlos Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. 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'13)*. ACM, New York, NY, 477–487.

Annie T. T. Ying and Martin P. Robillard. 2013. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, New York, NY, 655–658.

Annie T. T. Ying and Martin P. Robillard. 2014. Selection and presentation practices for code example summarization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. 460–471.

Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009—Object-Oriented Programming*. Lecture Notes in Computer Science, Vol. 5653. Springer, 318–343. DOI:http://dx.doi.org/10.1007/978-3-642-03013-0_15

Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE, Los Alamitos, CA, 563–572.

Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6, 429–445.

Received May 2015; revised June 2016; accepted August 2016