EMPIRICAL STUDIES OF ANDROID API USAGE: SUGGESTING RELATED API CALLS AND DETECTING LICENSE VIOLATIONS

Shams Abubakar Azad

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2015

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Shams Abubakar Azad**

Entitled: **Empirical Studies of Android API Usage: Suggesting Related API Calls and Detecting License Violations**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________________ Chair

Dr. V. Haarslev

______________________________ Examiner

Dr. J. Paquet

______________________________ Examiner

Dr. N. Tstantalis

______________________________ Supervisor

Dr. Peter C Rigby

Approved ______________________________

Chair of Department or Graduate Program Director

__________ 20 _____ ______________________________

Dean

Faculty of Engineering and Computer Science

Abstract

Empirical Studies of Android API Usage: Suggesting Related API Calls and Detecting License Violations

Shams Abubakar Azad

We mine the API method calls used by Android App developers to 1) suggest related API calls based on the version history of Apps, 2) suggest related API calls based on StackOverflow posts, and 3) find potential App copyright and license violations based the similarity of API calls made by them.

Zimmermann *et al* suggested that "Programmers who changed these functions also changed" functions that could be mined from previous groupings of functions found in the version history of a system. Our first contribution is to expand this approach to a community of Apps. Android developers use a set of API calls when creating Apps. These API methods are used in similar ways across multiple applications. Clustering co-changing API methods used by 230 Android Apps, we are able to predict the changes to API methods that individual App developers will make to their application with an average precision of 73% and recall of 25%.

Our second contribution can be characterized as "Programmers who discussed these functions were also interested in these functions." Informal discussion on StackOverflow provides a rich source of related API methods as developers provide solutions to common problems. Clustering salient API methods in the same highly ranked posts, we are able to create rules that predict the changes App developers will make with an average precision of 64% and recall of 15%.

Our last contribution is to find out whether proprietary Apps copy code from open source Apps, thereby violating the open source license. We have provided a set of techniques that determines how similar two Apps are based on the API calls they make. These techniques include android API calls matching, API calls coverage, App categories, Method/Class clusters and released size of Apps. To validate this approach we conduct a case study of 150 open source project and 950 proprietary projects.

Dedicated to my beloved Grandma

For her endless support, love, and encouragement.

Acknowledgments

First and foremost I offer my sincerest gratitude to my supervisor, Dr. Peter C Rigby for his advice, encouragement, motivation and immense knowledge. Without his guidance and support this work might not have been completed. Thank you for providing me with an excellent atmosphere and bringing out best in me. I could not have imagined having a better advisor and mentor for my master thesis.

I would like to thank Latifa for her guidance and support in this work and I am also very grateful to all the member of CESEL research lab for their selfless help.

I must also acknowledge Dr Tsantalis and Dr Paquet, Concordia University for there valuable suggestions and help.

I am very grateful to my family for everything I owe. I am indebted to my parents for all the scarifies they have made for me since the beginning. It was their love and motivation which helped me in moving forward in life

Finally I would like to thank all my friends especially Badal, Zahra, Vivek and Mitisha for their consistent help and encouragement. You guys are integral part of my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As of July 2013, Android had become the most dominant mobile operating system in the world and by that time more than 1 million of Android applications had been developed [67]. Further statistics show that during 2nd quarter of 2012 68% of smart phones shipped were Android based [34]. Again in one more survey 71% of developers reported to use this platform as their first choice [16]. These figures completely justify the exponential growth of Android software development and because of its promising growth it offers lucrative jobs to Android developers.

Android software developers use Application Programming Interfaces (APIs) to interact with libraries and frameworks with the aim of reducing the cost of development and increasing the quality of the product. There are many obstacles faced by developers when learning a new API, the most severe ones pertain to learning resources including documentation and code examples, as well as API structure such as its design or the name of its API elements [50].

To help developers learn and re-learn APIs, we have developed approaches that apply data mining techniques to determine API methods that are commonly used together. Like Zimmerman *et al* [73], we use the history of changes to the system to create association rules for methods that change together. However, instead of predicting the methods that belong to each individual application, as Zimmerman did, we predict the changes in the use of API methods that developers make to Android API method calls.

Android App development is time consuming and challenging. Generally a developer faces lots of competition from others and tries to launch his App on the market as soon as possible. To develop their App at much faster rate, proprietary developers may copy code from online resources. This copying may violate the license of open-source Apps. Unfortunately, we do not have the source code for proprietary Apps, so it is difficult to look for copying. We can reverse engineer the binary, however, many of the method names will be obfuscated. Fortunately, the names of Android APIs method calls remain unobfuscated so, our next goal is to use this information to determine if closed Apps are copying code from open source Apps.

This thesis focuses on two works. First, we want to develop a recommendation model that can recommend relatedness based on Android API call usage for Android developer which can help them during Android software development. Second, we want to develop a tool to find the Android Apps' license violations among different Android Apps.

1.1 Outline of Thesis

The thesis is organized in a way that each work gets separate space for the literature, research questions, methodology and data, and outcomes. This separation can be justified as both works have distinct goals, different approaches and outcomes. The last chapter combines and concludes these works.

1.1.1 Using Documentation and Version Control Histories to Suggest Related API Calls - Chapter 2

Research Questions: Can we predict which API method class will be changed together in the next version of an App based on the previous changes made to other Apps in the App Store? Can we predict which API will be changed in the next version of an App based on the API discussed on StackOverflow? Does the combined model (AppStore and StackOverflow models) have more predictive power ?

Data: We examine the Git repository of Android Apps and StackOverflow data to help developer to find the next likely to be changed Android API.

Literature: We will discuss it in chapter 2.

Methodology: Mine the Git repository of 250 Android Apps and 6 years of Stackoverflow data from 2008 to 2014 to find the association rules based on APIs changed together in Apps and API discussed together on Stackoverflow.

Outcome: Version history mining of different Apps and discussion post from Stackoverflow provides useful information to software developer for the development process of Apps. Our approach is able to predict the likely to be changed Android API with a precision of 73% and recall of 27%.

1.1.2 oftware metrics to suggest potential license violations. - Chapter 3

Research Questions: Which proprietary Android Apps are violating other open-source Android Apps? What portion of Android APIs from an open-source Apps is being copied by proprietary Android Apps ?

Data: Examine the Android APIs from proprietary Android Apps and open-source Android Apps to find the Apps' license violations among them.

Literature: We will discuss it in chapter 3.

Methodology: Downloaded 150 open-source Apps APK from F-Droid and 950 Apps APK from Google play store. We calculated the similarity between two application based on the number of public Android APIs they shared. We removed all the Android APIs that were present in more than 30% of Android Apps. The more an App is similar to the other, the more is the chances that it is copying the other one.

Outcome: We found that Apps having high overlap tends to copy more and Apps that fall in same category (as describe on Google play store) are much similar to each other. We also performed a manual analysis to determine if copying was indeed present. For legal reasons, we were not able to send the result to App developers to understand if they felt copying occurred.

1.1.3 Conclusion - Chapter 4

The main contribution of the first part of this research work is to create a recommendation model that can help software developer in the development process of a software by providing him/her important recommendations about the relatedness of API calls. The second part contributes in creating a tool that can help open-source App developer to check whether any proprietary Apps is violating license of his open-source App.

Chapter 2

Using Documentation and Version Control Histories to Suggest Related API Calls

Software developers use Application Programming Interfaces (APIs) to interact with libraries and frameworks with the aim of reducing the cost of development and increasing the quality of the product. There are many obstacles faced by developers when learning a new API, the most severe ones pertain to learning resources including documentation and code examples, as well as API structure such as its design or the usage of its API elements [50].

Furthermore, APIs evolve and developers must learn the new sequence of method calls to achieve a desired behavior. To help developers learn and re-learn APIs method usage, we have developed approaches that apply data mining techniques to determine

API method calls that are commonly used together. We extract the related APIs method calls from both development history, research approach 1 (RA1), and informal StackOverflow documentation (RA2). To evaluate our approaches, we predict the changes in API method calls that developers will make to their Apps. In total we examine 230 randomly-sampled Android Apps, 12k version of those Apps, and 152k API method calls. With version history, our best approach has an average precision of 70 percent and a recall of 27 percent. With informal API documentation, our best approach achieves a precision of 63 percent and a recall of 14 percent.

2.1 Research Approaches

RA 1, Version history: Using the version history of Apps we extract association rules of associated API method calls. We use the following approaches:

RA 1.1, Individual App Baseline: Like Zimmerman *et al* [73], we use the history of changes to the system to create rules for methods that change together. However, instead of predicting the methods that belong to each individual application, as Zimmerman did, we predict the changes that developers make to Android API method calls.

RA 1.2, Community of Apps: Using a single App, we are only able to create rules from the changes that have occurred in its past. Since the Android API is used in a similar manner across Apps, we are able 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 features that they may have never used before.

RA 1.3, Similar Apps: The Android API allows for a wide variety of applications that serve many different purposes from business to games. Similar to Gorla et al. [32] idea, we categorize the Android Apps based on their application type. However Gorla et al used Latent Dirichlet Allocation to categorized the Apps based on their description, we used application type described on Google play store. Apps from the same categories are expected to have common API calls. This will help us generating rules based on common API clusters and might increase the quality of our predictions.

RA 2, Informal Documentation on StackOverflow: Informal API documentation describes how to combine API methods to solve problems that developers commonly face. We generate association rules from the API methods present in high quality answer posts. These rules are then used to predict the changes to individual Android Apps. We have three approaches:

RA 2.1 StackOverflow Posts: To extract API methods from StackOverflow posts, we use Rigby and Robillard's tool that can acurately identify qualified API methods (e.g. Intent.addCategory) from natural freeform text and code snippets that don't necessarily compile [49]. We cluster all API methods present in highly rated answer posts.

RA 2.2 StackOverflow Code Snippets: Code snippets most often demonstrate the usage of an API [71, 70]. Recently, there has been much work on extracting code snippets to enhance traditional API documentation with up-to-date source code examples [61] as well as their summarization for better presenting code examples. We evaluate how well code snippets in answer posts predict the actual changes App developers make.

RA 2.2, Salient Methods on StackOverflow: Rigby and Robillard found code snippets contain setup code that is repeated across many posts [49]. For example, the API method *android.view.ViewGroup.findViewById* necessarily occurs every time with the method *android.view.LayoutInflater.inflate*. They found that code that was contained in free-form text tended to have a higher salience to the problem at hand. For example, API elements that are central to an example code fragment or have some discussion defining their function or describing their use. For this approach, we only consider API methods that are surrounded by natural language.

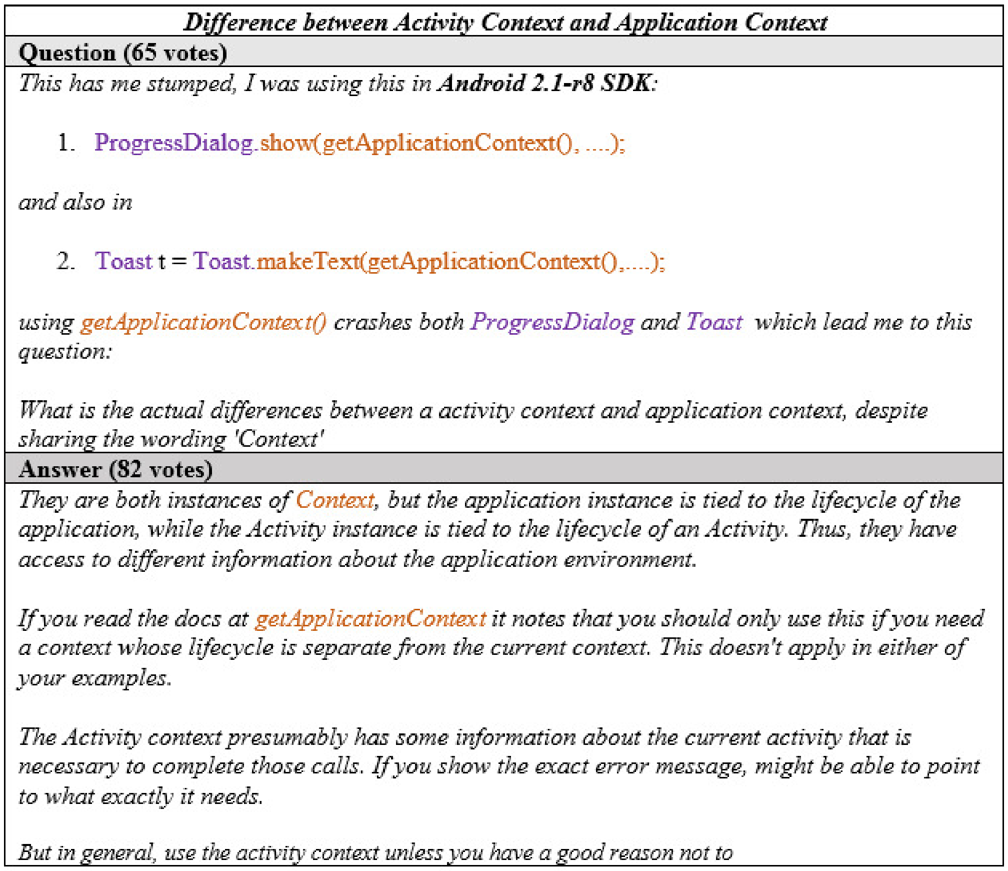
RA 3, Combining the best approaches: After evaluating the predictive power of each individual approach, we combine the top rules from the version history of a community of applications with the best StackOverflow rules. Our goal is to determine how complementary the set of rules are.

The remainder of this research work is structured as follows. In Section 2.2, we provide examples of related API elements in the App version history and in Stack-Overflow posts and code snippets. Section 2.3 describes our suggested approaches in detail. In Section 2.4, we evaluate how well each of our approaches predicts the changes that developers make to their Apps. In Section 2.6, we position this work in the literature. In Section 2.5, we discuss threats to validity. In Section 2.7, we conclude the research work and discuss the implications of our approaches.

2.2 Examples of relationships between API calls

In this section, we provide examples that illustrate relationships between API calls.

Fig. 1(b) represents the output of a *git-diff* showing an example of a set of Android API calls that have been changed together in a same commit in the Git repository of the Android SatNav App for GPS navigation. For example, the API methods *android.content.Intent.hasExtra* on line 40, *android.util.Log.e* on line 51, *android.os.Bundle.getInt* on line 57, and *android.content.res.Resources.getStringArray* on line 60 have been added together in the same change.



Difference between Activity Context and Application Context

Question (65 votes)

*This has me stumped, I was using this in **Android 2.1-r8 SDK**:*

1. ProgressDialog.show(getApplicationContext(), ...);

and also in

2. Toast t = Toast.makeText(getApplicationContext(),....);

using getApplicationContext() crashes both ProgressDialog and Toast which lead me to this question:

What is the actual differences between a activity context and application context, despite sharing the wording 'Context'

Answer (82 votes)

They are both instances of Context, but the application instance is tied to the lifecycle of the application, while the Activity instance is tied to the lifecycle of an Activity. Thus, they have access to different information about the application environment.

If you read the docs at getApplicationContext it notes that you should only use this if you need a context whose lifecycle is separate from the current context. This doesn't apply in either of your examples.

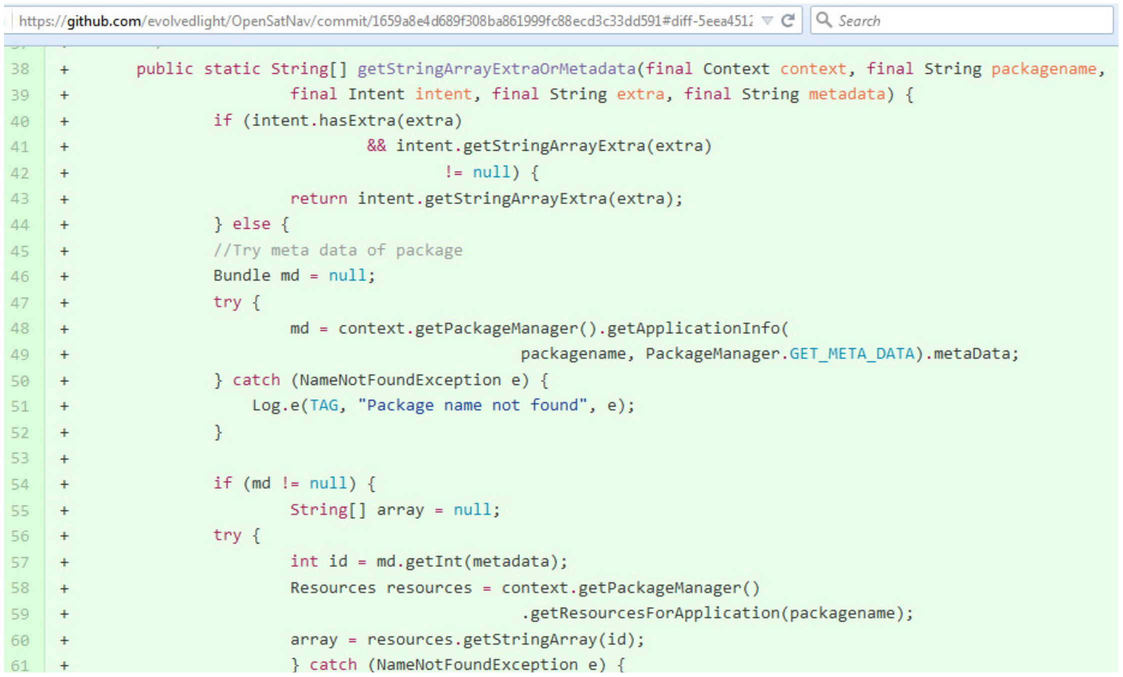
The Activity context presumably has some information about the current activity that is necessary to complete those calls. If you show the exact error message, might be able to point to what exactly it needs.

But in general, use the activity context unless you have a good reason not to

(a) Example of Android API methods discussed together on StackOverflow.

Professional developers also discuss Android API elements that perplex them on forums including StackOverflow. Fig. 1(a) shows an example of Android API methods, i.e. *ProgressDialog.show* on line 1 and *Toast.makeText* on line 2 discussed on highly-voted StackOverflow posts, i.e. question with 65 votes and answer having 85 votes, for the sake of understanding the differences between ActivityContext and ApplicationContext.

These examples motivate the use of both source code change history and documentation to discover relationships between API calls.



https://github.com/evolvedlight/OpenSatNav/commit/1659a8e4d689f308ba861999fc88ecd3c33dd591#diff-5eea451

```
+	public static String[] getStringArrayExtraOrMetadata(final Context context, final String packagename,
+			final Intent intent, final String extra, final String metadata) {
+		if (intent.hasExtra(extra)
+				&& intent.getStringArrayExtra(extra)
+					!= null) {
+			return intent.getStringArrayExtra(extra);
+		} else {
+		//Try meta data of package
+		Bundle md = null;
+		try {
+			md = context.getPackageManager().getApplicationInfo(
+					packagename, PackageManager.GET_META_DATA).metaData;
+		} catch (NameNotFoundException e) {
+		    Log.e(TAG, "Package name not found", e);
+		}
+
+		if (md != null) {
+			String[] array = null;
+		try {
+			int id = md.getInt(metadata);
+			Resources resources = context.getPackageManager()
+					.getResourcesForApplication(packagename);
+			array = resources.getStringArray(id);
+			} catch (NameNotFoundException e) {
```

(b) Example of Android API methods changed together in the Git version history.

2.2.1 Stage 1: Data Preprocessing

Before attempting to find patterns related to co-evolving API methods in development history and documentation, we need to ensure that the data is appropriate and that we have a reasonable size history to mine. In the following, we present the steps

Table 1: Characteristics of the Studied Apps.

Apps' Domains	Apps' Characteristics			
	#Apps	#Versions	#Changed Files	#API Methods 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
Puzzels	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

of data collection and data preprocessing that proceed the identification of atomic changes and clusters of discussed API elements.

The first step involved downloading the Git repositories of 230 free and open-source Android Apps. 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[1] catalogue. This catalogue contains 673 Android Apps in total. It provides information not only about the Git repositories of each Android application but also other information such as a link to the App

[1]https://f-droid.org/

on the Google Play Store.

We used random sampling to select Apps. 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 151k API elements, which limited the number of Apps we could study. Table 1 summarizes the main characteristics of the studied applications, including the App category (App Domain), 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), as well as the number of API methods changes (#API Methods Changes) per each category.

The second step consists of mining the Git repositories of each application to access information about the Apps evolution. 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 internal or external API element), the developer who committed the change, the author of the change, and the time of the change. The data was stored in PostgreSQL databases to facilitate further linking and processing.

2.2.2 Stage 2: Extracting API elements from Development History

Zimmermann *et al.* [73], focused on the code elements that made up the system, not the API calls. We focus exclusively on API method calls.

The process of identifying API elements is more difficult than that of identifying code elements internal to a system. To identify changing internal elements, one simply looks for changes inside a class to, for example, method declarations. The fully qualified name is apparent. However, with API method calls, one must resolve the type bindings to an external library.

There are two major challenges with identifying changes to API method calls. First, the identification of calls to APIs require advanced parsers that are able to resolve fully qualified names (FQN). Second, resolving this AST requires that one either builds each version of an App or extract a partial AST [20]. Therefore, we need an accurate approach to perform partial programming analysis since a simple Abstract Syntax Tree (AST) will fail to handle partial programs, resolve syntax ambiguities, and provide any type information when the declaration of a type is missing. There has been much excellent research into partial program analysis. For example, PARSEWeb is an approach that analyzes incomplete code, it performs type inference and resolves syntactic ambiguities. However, its prototype has not been fully evaluated and it is not publicly available. In addition, it is limited to two inference strategies only, i.e. the return type and method bindings [62]. Other parsers were designed for the same purpose including fuzzy parsers [30] which extract high-level structures out of incomplete or syntactically incorrect programs, and island grammars [43] which parse code snippets into islands grammer. Also, Gagnon *et al.* have proposed a technique to find declared types of local variables when starting from Java bytecode [17]. Their technique uses widely-adopted static analyzers that are based on a compiler framework that requires, in advance, the whole source code even when dealing with complete programs and type hierarchy, besides that there are no syntactic ambiguities in bytecode. Other techniques have relied on the use of static analyses to partially parse programs. Examples of such works include partial data flow and fragment analyses [15, 21, 52, 53] which require an Intermediate Representation (IR) generated from the complete code where no type declaration is missing. To overcome the above-mentioned shortcomings, we selected PPA (Partial Programming Analysis) developed by Dagenais and Hendren [12] to analyze partial programs at the level of each commit from the development history.

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. In some cases ambigiuity will remain, such as when the class and method name are known, but the package name is unknown. In experiments performed by the authors, the technique attained a precision of 92% [12].

PPA is not fast enough to process each file for each version, so we created a pipeline that allowed us to run PPA only on changed files and to extract the fully qualified name of methods 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 the removed code elements in the 'before' state and all the added ones in the 'after' state. If a code element occurred in both before and after, it was on a changed line, but remained unchanged itself. Finally, 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 also cover the API classes.

2.2.3 Stage 3: Identifying API methods in documentation

A great deal of knowledge about the behaviour of an API is contained in the informal discussions of developers as they help each other to solve problems. We group the API methods found in developer posts to predict the groupings of API methods that developers use in their Apps. These relationships should provide API method groupings that solve specific, recurring problems that App developers face and complement the co-evolving API method groupings found in the version history.

We extract qualified API methods (e.g. Class.method()) from StackOverflow. StackOverflow is a question and answer forum for professional developers [35][2]. 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.

Extracting API method 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 i.e. (less than 65 percent) [5]. In

[2] *http://developer.android.com/*

this investigation, we extract API method calls from each Android tagged StackOverflow post using that exist in the free-form text, i.e. StackOverflow posts using Rigby and Robillard's [49] Automatic Code Element extractor (ACE). ACE can extract code elements from documents that contain free-form text as well as code fragments that may not be compilable and handle large document collections with high precision and recall (above 0.90) [49]. ACE uses an island parser to identify code elements in documents. Unlike prior works [2, 36], this approach does not depend on an index of valid elements parsed from the source code of a particular system[13]. 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 that do not need an index of valid terms, can only parse code snippets and miss code elements that are in free-form text [61]. ACE performs the following stages in the code element identification process:

1. It uses an island parser to identify code-like terms from each document.
2. It creates an index of valid code elements based on step 1.
3. It re-parses each document to identify ambiguous terms that match code elements in the term index. It resolves each term using the term's context.
4. It outputs the code elements associated with each document.

Our goal is to provide developers with pertinent rules from documentation about API method that are related because repeatedly co-occurred in posts. We only include related API methods from the posts that are having at-least 1 upvote because these posts are acknowledged by the community to represent good solutions. We exclude questions 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 post does not show the relatedness of code elements so, it would introduce noise in our data and impact the accuracy of our predictions. We processed the StackOverflow posts tagged with 'android' from August 2008 till October 2014. There was a total of 22 million posts, including questions and lowly ranked answer posts. Of the 1 million highly voted posts we analyzed 495k that contained at least two API methods. In total we identified 2 million uses of API methods.

2.2.4 Stage 4: Identifying related API methods

In the previous stages, we described how we extracted API methods from the version history and from StackOverflow posts. In the former case, we group API methods based on those that are changed in the same commit. In the latter case, we group API methods based on those that co-occur in highly-voted answer posts. We can then generate rules from these groupings based on how frequently API methods co-occur. Using all groupings from the version history of all Apps will likely create a set of rules that are too general. As a result, we also group rules by application category.

Grouping APIs' Changes by Application Category

Recently, researchers have mined applications and clustered them by their description topics. They used as proxy for their implemented behaviour, the set of Android APIs that are used from within an application binary. The key idea was to associate descriptions and API usage to detect anomalies, i.e. applications whose behaviour would be unexpected given their descriptions [32]. Inspired from this study, we cluster applications by their category (i.e. communication, transport, finance, etc.) since similar applications are more likely to make calls to similar external APIs with the purpose of implementing similar behaviours. The aim is to show whether unlike previous works that investigated the source code change history of single projects [73?], we will be able to predict changes by mining development history across multiple similar projects.

We categorized the analyzed Android Apps based on the considered categories found in the Google Play Store[3].

'appid' is the common identifier used on F-Droid and Google Play Store. 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 since they were randomly-chosen from F-Droid (Cf. Table 1). We group historical changes by community of applications belonging to each category, then we generate recommendations concerning co-changing API methods using the development history of each cluster of similar applications, and finally we compute precision and recall for our recommendations.

[3]https://play.google.com/store?hl=en

[4]https://code.google.com/p/android-market-api/

We compare the precision and recall for the single App and community of Apps approaches.

Grouping Discussed API methods by Developers' Discussions

Prior to mining relationships between API methods from documentation, we needed to create clusters of discussed API elements from documentation. We used as a document units, highly-voted StackOverflow answers. We clustered the API methods mentioned in each highly-voted StackOverflow answer while ignoring stack-traces using appropriate PERL scripts. We considered only Android API methods, i.e. those belonging to the package *Android** since the context of our study consists of Android Apps.

We distinguish three investigations when dealing with API documentation. The first model leverages the entire content of each highly-voted StackOverflow answer, i.e. all mentioned API methods including the ones trapped in natural language text as well as the ones in code snippets. The second model exploits API methods in code snippets only, while the last approach focuses on salient methods, i.e. those that are mentioned in natural language text exclusively.

We dealt with a total of 141,629 StackOverflow answers posts in the first investigation while we analyzed 105,236 posts in the case of code snippets. Finally, we had a total of 36,393 posts containing salient methods (API methods discussed in post).

2.2.5 Step 5: Association Rule Mining

We mine rules relating API methods from both development histories and documentation. We followed the following procedure.

Grouping API Methods into Transactions

Before describing how our mining approach works, we first introduce the following notions: *transactions* and *items* as well as their definitions depending on the context in which they are used, i.e. change history or documentation.

When exploiting change history, we distinguish two different high-level changes: addition or removal of an API method. We focus on these two high-level changes because we are interested in calls to external APIs. An API method that has been changed in a particular commit is called an *item* in our case. A transaction consists of a set of atomic changes *items*–each of them represents a API method change–identified by the same author, same date, and message under Git.

When leveraging change documentation, we define a transaction as the set of API methods discussed together in the same high-quality voted answer in the StackOverflow where each API method represent an *item* of a transaction. Thus, a transaction is identified by the a developer, date, and content which correspond to the actual StackOverflow post in question. The sets of all transactions represent the input for the mining phase.

We address the following: *"I added this API method during my source code change task; which other methods are typically relevant to my task?"*

An example that illustrates the notion of *transactions* is as follows:

$$T = \left\{ \begin{array}{c} android.widget.TextView.setVisibility \\ android.app.ListActivity.findViewById \\ android.app.Activity.getApplicationContext \\ android.app.Activity.getWindow \\ android.app.Acctivity.onCreate \\ android.widget.Toast.makeText \end{array} \right\}$$

where T consists of six atomic changes underwent by the above-mentioned Android API methods involving additions of the methods *android.app.ListActivity.findViewById*, *android.app.Activity.getApplicationContext*, and *android.app.Acctivity.onCreate* as well as deletions of *android.widget.TextView.setVisibility*, *android.app.Activity.getWindow*, and *android.widget.Toast.makeTex*.

We filter transactions to eliminate those consisting of more than 100 API methods because as stated in previous works these long transactions do not usually correspond to meaningful atomic changes, such as feature requests and bug fixes [23]. An example is when developers perform an initial commit or a license change that need to be reflected in all files or when they refactor code such as the case of the Eclipse IDE where developers organize their java files' import declarations that involve a large number of files [**?**]. Additionally, we exclude transactions consisting of one item since such transactions cannot help when performing the prediction.

Our applications differ in their development history. For example the Android application *aGrep* belonging to the category *Tools* has five years of source code history, it has been under Git since 2010, whereas the Android application *abstract-art* from the category *Personalization* has only three years of development history. Each of these applications also involves several developers, reducing the likelihood that particular programming habits or practices of a specific developer considerably affects the accuracy of the predictability of the suggested approach. Additionally, the number of developers contributing to StackOverflow is also large alleviating possible threats related to transactions built from StackOverflow best answers.

From Transactions to Rules

Association rule mining is a means of discovering relationships between items of transactions in a database. We generate *rules* and show the extent to which these items are strongly related to each other based on the computation of probabilities (i.e. support and confidence).

Given a set of transactions from development history or documentation, the goal of our approach is to mine *rules* from these transactions and suggest to developers pertinent recommendations about co-evolving API elements. An example of rule mined from development history is as follows:

$$android.view.MenuItem.getItemId \Rightarrow android.view.LayoutInflater.inflate$$

This rule means that whenever the developer changes the Android API method *android.view.MenuItem.getItemId*, then she should also change the method *android.view.LayoutInflater.inflate.*

Formally, an association rule r: $X \Rightarrow Y$ is a pair (X,Y) of two disjoint itemsets X and Y where X is called the *antecedent* (if) and Y the *consequent* (then). Rules computed from data, unlike the if-then rules of logic, are probabilistic in nature. They are interpreted based on the amount of evidence, that is, the number of transactions a rule is derived from. Therefore, the implication in the rule is based on the knowledge discovered from learning history or documentation and should be not considered as an absolute truth.

Association rule mining discovers all rules in the data that satisfy a user-specified minimum support and minimum confidence. Minimum support represents the minimum amount of evidence, that is the number of transactions required to consider a rule valid and minimum confidence specifies how strong the implication of a rule must be to be considered valuable:

- **Support**. The support is defined as the number of transactions from which the rule has been derived. Assume that the API method call *android.view.MenuItem.getItemId* was changed in 30 transactions. Of these 30 transactions, 6 also involved changes of both the *android.view.LayoutInflater.inflate.* The *support* for the above rule is then equal to 6. Formally, the support is defined as follows: The *support* of a rule $X \Rightarrow Y$ by a set of transactions T is:

$$support(X \Rightarrow Y) = \frac{count(X \Rightarrow Y)}{|T|}$$

where

  - $count(X \Rightarrow Y)$ represents the number of transactions where $X \cup Y$ occurs together.

  - $|T|$ is the total number of transactions in a database.

- **Confidence**. The confidence reflects the strength of the rule, i.e. the consequence. It is the ratio of the number of transactions that contain all items in the consequent and antecedent (i.e. support) to the number of transactions including all items in a given antecedent. In the above example, the consequence of changing *android.view.MenuItem.getItemId* and *android.view.LayoutInflater.inflate* applies in 6 out of 30 transactions. Hence, the *confidence* for the above rule is 0.2.

  Formally, the confidence is defined as follows:

  The *confidence* of a rule $X \Rightarrow Y$ by a set of transactions T is:

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

  where

  - $support(X \Rightarrow Y)$ represents the support of a rule $X \Rightarrow Y$ by a set of transactions T.

  - T is the total number of transactions in the analyzed database.

Following Zimmermann *et al* [73], when we had a large number of transactions that reduced the support values of rules, we used the *support count*. The support count is formally defined as follows:

- **Support count**. The support count determines the number of transactions the rule has been derived from. Assume that the method has changed in 20 transactions. Of these 20 transactions, 10 also included changes of the method. Therefore, the support count for the above rule is 10.

  The *support count* of a rule $X \Rightarrow Y$ by a set of transactions T is:

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

  where

  - $frequency(X \cup Y)$ represents the number of transactions where $X \cup Y$ occurs together.

  Consequently, the *confidence in case of support count* of a rule $X \Rightarrow Y$ by a set of transactions T is as follows:

  $$confidence(X \Rightarrow Y) = \frac{support\_count(X \Rightarrow Y)}{support\_count(X)}$$

  where

  - $support\_count(X \Rightarrow Y)$ represents the support count of a rule $X \Rightarrow Y$ by a set of transactions T.
  - $support\_count(X)$ is the number of transactions containing X.

We generate rules from two databases: the first database consists of transactions built by mining development history across multiple similar projects while the second database consists of transactions created by mining documentation history. The association rule mining algorithm used in the mining phase is the classical approach widely applied in data mining namely, i.e. the *Apriori* algorithm.

Apriori

The classical approach to compute association rules is the *Apriori Algorithm* [1]. The Apriori algorithm takes as input a minimum support and a minimum confidence and computes the set of all association rules. The support measure helps to reduce the number of candidate itemsets explored during the frequent itemset generation phase. The pruning of candidate itemsets using support is guided by the Apriori principle: *if an itemset is frequent, then all of its subsets must also be frequent* [1].

The traditional and simple way of applying the Apriori algorithm is to compute *all rules* beforehand and then identify the rules corresponding to a given item. However, computing all possible rules can be time demanding–up to 2-3 days in our experiments since, our approach tries to optimize the computation time by bringing the following modifications to the mining algorithm when generating the association rules:

An association rule consist of two part, the left hand side of it is known as antecedent and the right hand side of it is known as consequent.

$antecedent \rightarrow consequent$

- **Single antecedents.** We consider only rules with a single antecedent. Thus, our approach computes only rules with a single item in their antecedents. Association rules with more than one item in their antecedents such as api_1, api_2 $\Rightarrow api_3$ are therefore not considered.

- **Single consequents.** We have modified the approach such that it only computes rules with a single item in their consequent. So, for a given item (e.g. *api1*); the rules have the form $api_1 \Rightarrow api_2$. As shown in previous works, rules with single consequents are sufficient when considering the union of all the consequents for a given item [73].

The above-mentioned considerations reduced the computation time of our approach whose core part is the mining phase which uses a set of PostgreSQL database queries, Perl and R scripts to 1) generate atomic change sets and, thus, transactions from the source code change history of (individual, all, or similar) applications, 2) build clusters of discussed API elements which form the transactions for the documentation, 3) access the transactions per individual or clusters of (all or similar) applications (when dealing with development history) and posts (when leveraging documentation) plus 4) take into account, using R scripts, the developer-specified minimum support and confidence thresholds to compute rules and report their support and confidence values.

The average runtime of the approach varies with the amount of analyzed history. When dealing for example with large version histories (i.e. thousands of transactions) such as the case for the categories *Tools* having 3,485 transactions and *Communication* consisting of 1,295 ones, the computation time is about 15 minutes, measured on a standard workstation Intel Core i5-2400 CPU 3.10 GHZ and 12 GB RAM. The partial programming analysis phase time is not included here as it was used to generate FQN for each code element but not for finding rules. However, after initially parsing the changes, which is very time consuming because there was over 150k changes, the parsing of each change in real time will not be time consuming and can easily be appended in the database.

Analyzing the entire StackOverflow and identifying API elements present in it using the ACE approach took us almost 4 days of computation time which is reasonable given the large amount of discussions (i.e. 22 millions of posts). Predicting co-evolving API elements using new discussions posted in StackOverflow will require us to update our database with the new information.

Generating and Filtering Rules

Assume a developer has performed some changes, following Zimmermann *et al.* [73], we refer to the set of items that underwent changes as the *situation* S. An example of situation is as follows:

$$S = \{android.app.Activity.getRessources\}$$

Given a situation, our approach predicts likely changes in API elements by considering pattern matching rules. A rule *matches* an item if it is equal to the antecedent of the rule. Our approach suggests a list of API elements L for a situation S and a set of rules R by considering the *union* of the consequents of all matching rules:

$$A = \cup_{(S \Rightarrow x) \in R} x$$

We rank the matching rules by confidence. Thus, for all rules having the same antecedent, the union of the consequents of all rules is assigned. We choose the confidence as a criterion for ordering rules because by definition the confidence measure reflects the pertinence and strength of a rule. The number of generated rules varies with the specified support and confidence thresholds which are input parameters in our approach. Low thresholds such as 0.001 and 0.1 would overwhelm developers with suggestions. As a result, we show only the top 10 rules R_{10} ranked by confidence instead of the entire set R.

$$R_{10} \subset R$$

The notions of *situation*, *matching rules*, and *top 10* rules used by our approach applies for both rules generated from development history as well as documentation, i.e. StackOverflow.

2.3 Rule Examples

In the following, we present an illustrative example of actual rules mined from development history and StackOverflow.

Using the **development history** we find that the API methods *android.content.Context.getResources* and *android.content.Context.getString* occur in the same change sets. These two APIs have been used by 27 Android Apps and have been changed together in 47 transactions. In a set of these 47 transactions, this rule was triggered by a change in the type:

$$\left\{\begin{matrix} android.content.Context.getResources \Rightarrow \\ android.content.Context.getString \\ android.content.Context.createDisplayContext \end{matrix}\right\}_{[7;10]}$$

This means that whenever the API method *android.content.Context.getResources* changes, the *android.content.Context.getString* should change as well. The minimum support and confidence used to generate this rule are respectively 0.01 and 0.5.

On the highly voted **StackOverflow answer posts**, we find that the API methods the API methods *SQLiteDatabase.update* and *SQLiteDatabase.insert* have been discussed together in 6 transactions. An example of rule mined is as follows:

$$\left\{\begin{matrix} Environment.getExternalStorageDirectory \Rightarrow \\ android.content.Context.getFilesDir \\ android.content.Context.openFileOuput \end{matrix}\right\}_{[5;70]}$$

Whenever a developer changes the API method *Context.getFilesDir*, he or she should be aware of the method *Environment.getExternalStorageDirectory* since they are often discussed together on StackOverflow.

2.4 Empirical Evaluation

2.4.1 Evaluation Setup and Analysis Method

The validation process required dividing our data into *training* and *test data*. The *training data (T_r)* was used to predict related API methods that were then used to suggest API methods co-evolution rules for the *test data (T_s)* set. Thus, we analyzed for each set of transactions T from the test set whether its items ($items(T)$) can be *predicted* from the training set.

We arranged all the transactions in accending order of date of commit or accending order of date of post and dividev them into two parts. The training data set consists of 80 percent of the transactions, while the test set consists of 20 percent of recent data for each App. All of the transactions in the training set are older than the test set. Since some Apps are small, we ensured that each division had at least one full month of test data.

The procedure followed during our evaluation process can be summarized as follows:

1. Based on the test set, i.e. T_s, we prepared a number of *queries* for each transaction. A query $q = (Q, W)$ consists of a *query* $Q \subset items(T)$ and an expected *outcome* $W = items(T) - Q$. For each App, for each transaction T with $|T| >= 2$ and each item $m \in$ T from its evaluation period, we consider the situation $Q = m$ and check whether the approach would predict the expected outcome $W = T - m$. We dealt with $|T|$ queries per transaction T, whose items

each consists of a single antecedent $t \in T$.

2. For each query $q = (Q, W)$, we consider all transactions T_r that have been completed before time(T) as a *training* set and mine the set of association rules R from these transactions with respect to Q.

3. The number of generated rules can be huge (depending on the support and confidence thresholds). From a tool perspective, the developer who will use this approach does not have to be overwhelmed by endless lists of API methods change suggestions. For such a purpose, we consider the *top 10 single-consequent rules* $R_{10} \subset R$ ordered by confidence.

4. The result $M_q(R_{10})$ of a query $q = (Q, W)$ consists of two parts:

    - $M_q \cap W_q$ consists of items that *matched* the expected result and, therefore, are considered *correct*. W_q is the expected outcome for the query q.
    - M_q - W_q are unexpected rules which are *incorrect*.

For the assessment of a result M_q for a query $q = (Q, W)$, we use three measures from information retrieval [65]: the *precision* P_q shows the percentage of API elements that correspond to the expected outcome. The *recall* R_q indicates the percentage of expected API methods changes that were returned. To provide an aggregated, overall measure of precision and recall, we use the F-Measure F_q, which is the harmonic mean of precision and recall:

$$P_q = \frac{|M_q \cap W_q|}{|M_q|}$$

$$R_q = \frac{|M_q \cap W_q|}{|W_q|}$$

$$F_q = \frac{2.P_q.R_q}{P_q + R_q}$$

We compute for each query q, the triple (P_q, R_q, F_q) of precision, recall, and F-measure. To measure the overall performance of the approach for all assessed queries A generated from transactions of the evaluation period, i.e. those belonging to the test set (T_s), we summarize the obtained triples of precision, recall, and F-measure into single triple based on a macroevolution technique from information retrieval. Macroevolution computes the average values of precision, recall, and F-measure triples of the queries A:

$$A^* = \{q | q = (Q, W) \in A, M_q \neq \emptyset\}$$

If the approach does not return any API methods change suggestions for a query q (that is, $Mq = \emptyset$), we exclude such queries from our analysis to avoid impacting the accuracy and more specifically the approach's precision. Thus, unless otherwise noted, our analysis includes only the queries A^* where Mq is not empty:

$$P_M = \frac{1}{|A^*|} \sum_{q \in A^*} P_q$$

$$R_M = \frac{1}{|A^*|} \sum_{q \in A^*} R_q$$

$$F_M = \frac{1}{|A^*|} \sum_{q \in A^*} F_q$$

where P_M, R_M, and F_M are respectively the averages of precision, recall, and F-measure over all queries of each single application. The averages of these values over all studied Apps represent the final precision $P_{M'}$, recall $R_{M'}$, and F-measure $F_{M'}$ of each suggested approach.

2.4.2 Comparing Approaches

To compare the predictive power of each approach, we conducted pair-wise comparisons of the precision, recall, and F-measure using a non-parametric test for pair-wise median comparison, specifically the Wilcoxon paired test. We chose a paired test because our samples are dependent, as we compute, for each individual App, its corresponding precision, recall, and F-measure by applying the different approaches.

The Wilcoxon test indicates whether the median difference between two approaches is significantly different from zero i.e. $H_0 : \mu_d = 0$, where μ_d is the median of the differences.

Since we execute the Wilcoxon paired test multiple times to compare the predictive power of the various approaches, we must correct significant p-values. We use the Holm correction [24], which is similar to the Bonferroni correction, but less stringent. It works as follows: (i) the p-values obtained from multiple tests are ranked from the smallest to the largest, (ii) the first p-value is multiplied by the number of tests performed (n), and is deemed to be significant if it is less than 0.05, and (iii) the second p-value is multiplied by $n - 1$, and so on. In Table 2, we have shown the result of Wilcoxon paired test that shows our approach is better than the baseline.

Table 2: Comparison among Approaches: Results of Wilcoxon Paired Test

Precision		
Approach 1	Approach 2	Adjusted p-value
Basline (Individual Apps)	All Apps	**<0.001**
All Apps	Similar Apps	**<0.001**
StackOverflow	StackOverflow Code Snippets	**0.182**
StackOverflow Code Snippets	Salient Methods	**0.025**
Baseline (Individual Apps)	Development History and StackOverflow	**<0.001**
Recall		
Baseline (Individual Apps)	All Apps	**0.64**
All Apps	Similar Apps	**0.0139**
Stackoverflow	StackOverflow Code Snippets	**0.9798**
StackOverflow Code Snippets	Salient Methods	**<0.001**
Baseline (Individual Apps)	Development History and StackOverflow	**0.059**
F-measure		
Baseline (Individual Apps)	All Apps	**<0.001**
All Apps	Similar Apps	**0.1174**
StackOverflow	StackOverflow Code Snippets	**0.9714**
StackOverflow Code Snippets	Salient Methods	**<0.001**
Baseline (Individual Apps)	Development History and StackOverflow	**<0.001**

2.4.3 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 methods that change together using association mining rules. We have three approaches. First, we replicate past work by making predictions using single Apps. Second, we combine the rules generated across the entire community of Apps. Third, we combine rules that come from similar Apps, such as Apps in a similar domain. In all cases, we divide the data into a training and test set, grouping co-changing API methods to create rules in the training set and predicting co-changing API methods in the test set.

RA 1.1: Individual App Baseline

Our baseline leverages source code changes of individual Apps to predict changes in API methods for each app.

Table 3: Predictability Results using Development History Mined for Individual Apps: $P_{M'}$ = Precision , $R_{M'}$ = Recall, $F_{M'}$ = F-measure.

	Development History Results				
Metrics	*1Q*	*Median*	*Mean*	*3Q*	*Max*
$P_{M'}$	17.79	28.75	**35.60**	48.39	100
$R_{M'}$	8.65	18.06	**22.06**	29.66	100
$F_{M'}$	12.97	19.81	**21.27**	28.68	57.77

The training set of our baseline consists of a total of 9,266 transactions and 51,908 items while the evaluation test deals with 2,520 transactions and 12,521 items in total. We had, on average, 27 transactions per App for the training part while we dealt with 8 transactions, on average, from the test set. Each transaction consists of, an average, of 6 items.

Since we had to deal with the development history for 230 different individual Apps, we had to perform several experiments with different minimum support and confidence thresholds for each App. To facilitate comparison across all applications, we decided on a common value for all Apps as in previous works [73]. We chose thresholds that are not too low and not too high to ensure a trade-off between the number of rules and their relevance. Table 3 reports the descriptive statistics of the precision, recall, and F-measure obtained with a support count threshold equal to 5 and a confidence threshold equal to 60 percent.

The findings from Table 3 indicate that our single App version history baseline approach is able to predict API method changes with an average precision of 36 percent, recall of 22 percent, and F-measure of 21 percent. While relatively low, these results are consistent with Zimmermann *et al.* [73] 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 Zimmermann *et al.* who used large systems like Eclipse and JBoss. We suspect that one reason we achieved better precision (7 points higher) and lower recall (22 points lower) is because we are predicting API method calls and not internal method changes. The possible set of rules is smaller with API methods, so our predictions may be accurate despite a short version history.

RA 1.2: Community of Apps

Our baseline produced a reasonable precision and recall despite a short version history. Since API methods are used in similar patterns across multiple Apps, we use the version history of all Apps to create rules to predict the changes that will be made to individual Apps.

We dealt with a training period of six years going from 2007 to 2013, we had in total 9,738 transactions and 120,401 items. Our evaluation period consists of one year, it goes from 2013 to 2014 and consists of 2,435 transactions and 30,223 items to be evaluated from the test set. Similarly to our first investigation, we experimentally determined the minimum support and confidence thresholds suitable for our training set by means of several experiments starting from 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

Table 4: Predictability Results using Source Code Changes Mined across All Apps: $P_{M'}$ = Precision , $R_{M'}$ = Recall, $F_{M'}$ = F-measure.

	Development History Results				
Metrics	*1Q*	*Median*	*Mean*	*3Q*	*Max*
$P_{M'}$	64.27	73.64	**75.14**	87.84	100
$R_{M'}$	9.74	17.36	**22.32**	29.07	100
$F_{M'}$	17.39	28.74	**30.57**	41.16	93.72

configuration chosen consists of a support count of 12 and a confidence of 70 percent.

Table 4 summarizes the results of investigating the source code change history across all Apps. Results indicate that we can predict changes in individual Apps with an average precision of 75 percent and an average recall of 22 percent. Compared to our baseline, we increase our precision by 40 points, and recall remain almost same. The difference in terms of precision is statistically significant with a p-value <0.001 while there is no statistically significant difference in terms of recall. Consequently, the difference in terms of F-measure is statically significant with a p-value<0.001.

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

The low recall, indicates that there are many different ways to combine API methods, and while we accurately suggest related API methods, we miss many of the possible combinations. Since we are only suggesting the top 10 related API methods,

Table 5: Development History of Analyzed Projects (Txn = Transaction).

Apps' Domains	Data from Source Code		
Category	#Files Changed	#Txns	#Items
Arcade	37	39	115
Books & Reference	495	396	1,310
Business	40	62	254
Communication	1,151	1,295	6,719
Education	203	337	1,247
Entertainment	457	560	1,882
Finance	658	566	3,590
Health & Fitness	35	57	317
Libraries & Demo	24	17	164
Lifestyle	48	47	231
Media & Video	694	653	2,635
Music & Audio	295	338	1,379
News & Magazines	100	160	1,107
Personalization	440	616	3,389
Photography	22	11	92
Productivity	502	329	2,513
Puzzle	84	71	201
Social	286	338	1,773
Tools	3,844	3,485	21,948
Transportation	107	144	585
Travel & Local	86	97	402

we often miss methods that developer 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 from a Traffic App.

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 5, we show the 22 App clusters, from business to Acarde to Travel & Local.

Table 6: Evaluation Periods (Txn = Transaction) for Development History.

Apps' Domains	Data from Source Code		
Category	Evaluation Period	#Txns	#Items
Arcade	2012-11-26 to 2013-01-09	10	20
Books & Reference	2013-06-07 to 2014-07-01	101	263
Business	2014-03-10 to 2014-07-03	16	69
Communication	2013-12-02 to 2014-06-25	327	1,289
Education	2012-03-07 to 2014-04-16	88	291
Entertainment	2014-01-06 to 2014-07-03	145	794
Finance	2013-12-29 to 2014-07-02	144	946
Health & Fitness	2011-02-24 to 2014-03-30	16	53
Libraries & Demo	2014-04-11 to 2014-05-11	5	10
Lifestyle	2012-11-17 to 2013-12-11	14	39
Media & Video	2013-08-31 to 2014-06-30	167	761
Music & Audio	2013-10-21 to 2014-05-19	88	419
News & Magazines	2013-12-27 to 2014-05-22	41	178
Personalization	2012-07-07 to 2013-04-14	159	657
Photography	2012-08-02 to 2012-08-18	4	34
Productivity	2013-10-12 to 2014-06-25	90	361
Puzzle	2012-02-27 to 2014-04-22	19	128
Social	2013-12-27 to 2014-06-27	88	328
Tools	2013-10-23 to 2014-07-04	929	5,572
Transportation	2013-09-13 to 2014-01-28	41	177
Travel & Local	2013-10-22 to 2014-04-26	26	125

Our training set consists of a total of 9,266 transactions and 51,908 items; on an average there are 458 transactions and 2469 items per category. See Table 5 for more details.Since each category consists of several Apps, the date shown in the first column (Column *Apps in Git since*) is the date since which the source code of the oldest App from each category has been made under Git. In addition, it indicates the number of source code files changed, the number of transactions, as well as the number of items per each category of Apps. The evaluation period consists of 2,520 transactions

Table 7: Predictability Results using Source Code Changes Mined Across Multiple Apps: $P_{M'}$ = Precision , $R_{M'}$ = Recall, $F_{M'}$ = F-measure.

	Development History Results				
Metrics	*1Q*	*Median*	*Mean*	*3Q*	*Max*
$P_{M'}$	56.80	70.52	**70.02**	87.27	100
$R_{M'}$	9.07	19.83	**27.43**	38.58	100
$F_{M'}$	15.50	30.25	**33.79**	49.43	98.93

and 12,521 items in total; it included the analysis of on average 120 transactions and 596 items per category from the test set. For space reasons, we report in Table 6, the evaluation periods corresponding to each category (Column *Evaluation Period*) instead of single Apps.

Since we had a development history of 22 different categories, we had to perform several experiments with different minimum support and confidence thresholds for each specific category to select the appropriate minimum support and confidence thresholds whose values are in our case 10 and 50 percents respectively. Table 7 summarizes the findings obtained using the notion of similar Apps and which can be interpreted as follows.

Leveraging source code changes across similar Apps we find that we can predict changes in individual Apps with an average precision of 70 percent and an average recall of 27 percent. Compared to our baseline, we increase our precision by 35 and we also improve our recall by 6 points. The difference in terms of precision is statistically significant with a p-value$<$ **0.001** while there is no statistically significant result in terms of recall as well as F-measure.

The high increase in precision clearly shows that developers make use of the same API methods to implement similar behaviors for similar Apps. The slight increase in recall, even though not statistically significant, is likely due to the increase in the amount of investigated history. In effect, while the baseline leverages development history of single Apps, the similar Apps-based approach exploits a larger search space consisting of the source code change history mined across all Apps similar to a particular App in question.

It is important to mention that since we randomly sampled our Apps, clusters of similar Apps may be different in terms of their size and thus the amount of their development history. Illustrative examples are Apps from the categories *Tools* and *Productivity* which vary in their size. In fact, Apps from the category *Tools* leverage source code change history from a cluster of 120 Apps consisting of 56,930 API methods' changes while Apps from the category *Productivity* predict changes in API methods using source code changes mined across 14 Apps having a total of 6,024 APIs methods' changes. In general, we observed from our experiments that when leveraging large clusters of similar Apps, our Similar Apps-based approach yield better prediction results since its benefit from a larger amount of exploited learning history. This finding is inline with the statement by Zimmermann *et al.* [73]: *the more there is to learn from history, the more and better change suggestions can be made.*

2.4.4 RA 2: Informal API Documentation

Informal API documentation contains rich information about APIs used by Apps [71, 61, 49]. On popular forums such as StackOverflow, 22 millions of posts mention API elements. Therefore, we leverage Android StackOverflow to discover related API methods.

RA 2.1: StackOverflow Posts

We propose as a first approach to leverage the entire content of highly-voted answers from StackOverflow. Our training set consist of all clusters of API methods mentioned together in highly-voted StackOverflow answers posts. Specifically, we had a total of 113,303 transactions and 406,622 items from our training set while our test set consists of a total of 2,520 transactions and 12,521 items. We have experimentally tried different support and confidence thresholds prior to choosing final values. In general low support values help finding more rules but which are not necessary pertinent. Thus, we have chosen values that are not that much low or high to avoid impacting the precision of the approach. Our final setting consists of a support count of 8 and a confidence of 70 percent. Table 8 reports the results obtained with our StackOverflow-based approach and which can interpreted as follows.

Using API documentation we find that we can predict changes in individual Apps with an average precision of 66 percent and an average recall of 12 percent. Compared to our baseline, we increase our precision by 31 points, but decrease our recall by 10 points.

Table 8: Results of Change Predictability using Informal API Documentation (StackOverflow) : $P_{M'}$ = Precision , $R_{M'}$ = Recall, $F_{M'}$ = F-measure.

StackOverflow Results					
Metrics	*1Q*	*Median*	*Mean*	*3Q*	*Max*
$P_{M'}$	39.90	57.09	**65.57**	96.18	100
$R_{M'}$	3.64	6.73	**12.65**	13.22	58.11
$F_{M'}$	6.86	11.79	**16.01**	18.66	94.30

The high increase in precision is expected since we are investigating only highly-voted StackOverflow answers posts which, as shown by previous works [33], reflect changes in Android APIs. The very small decrease in recall can be justified by the fact that not all the changes made to source code and in particular API methods are reflected in informal developers' discussions. Furthermore, the patterns concluded from informal API documentation are more oriented usage and thus it is not counter-intuitive to have few cases were it would be impossible to predict changes for API methods using StackOverflow.

Overall, we conclude that informal API documentation, in particular StackOverflow, can be used to complement approaches based on development history when discovering related API methods.

RA 2.2: StackOverflow Code Snippets

Code snippets are an important source for answering questions about software libraries and applications, they are, usually, used to illustrate the usage of API elements, or to remind developers of known idiom [71, 61]. Recently, researchers have shown that 65 percent of accepted answers on StackOverflow contain code examples [60], while unanswered questions often lack code [3]. To show whether code snippets help discover relationships between API calls, we suggest an approach that leverages code snippets present in highly-voted StackOverflow answers posts.

Our training set consists of all clusters of API methods present in code snippets trapped in StackOverflow answers posts. We had, in total, 84,189 transactions and 361,884 items while our test set remains the same, i.e. it consists of the 2,520 transactions and 2,521 items evaluated by all other approaches. After preforming several experiments using our training set, we selected as our final setting a minimum support count of 10 and a minimum confidence of 70 percent.

Table 9 reports the findings of our code snippets-based approach which we can interpret as follows.

Using code snippets we find that we can predict changes in individual Apps with an average precision of 62 percent and an average recall of 14 percent. Compared to our baseline, we increase our precision by 28 points, but decrease our recall by 8 points.

Not surprisingly, there is a high increase in precision which is likely due to the pertinence of the relationship between API elements present in code snippets, which

Table 9: Results of Change Predictability using Code Snippets on StackOverflow: $P_{M'}$ = Precision , $R_{M'}$ = Recall, $F_{M'}$ = F-measure.

StackOverflow Results					
Metrics	*1Q*	*Median*	*Mean*	*3Q*	*Max*
$P_{M'}$	37.96	65.94	**62.98**	89.27	100
$R_{M'}$	3.25	6.32	**14.06**	11.77	100
$F_{M'}$	5.94	10.96	**16.87**	16.27	100

are most often used to illustrate and describe a well-focused problem at hand. In fact, recent research has shown that when a programmer searches for information related to an API, of the various kinds of documentation he/she finds on the Web, code examples are one of the most effective [37], important [51], and frequently sought-after [46]. The importance of code examples has recently lead to new emerging research directions which focus on extracting code examples found in documentation [61] as well as their summarization [71, 70].

The slight decrease obtained in recall can be justified by the fact that we miss some API methods mentioned in the natural descriptive text of StackOverflow posts, and which are most often cental to code snippets [49].

Overall, we conclude that code examples can help predicting relationships between API methods when performing software change tasks.

Table 10: Results of Change Predictability using Salient Methods on StackOverflow: $P_{M'}$ = Precision , $R_{M'}$ = Recall, $F_{M'}$ = F-measure.

StackOverflow Results					
Metrics	*1Q*	*Median*	*Mean*	*3Q*	*Max*
$P_{M'}$	36.78	64.18	**61.56**	87.15	100
$R_{M'}$	4.196	7.49	**17.00**	16.79	100
$F_{M'}$	7.75	12.71	**19.55**	23.35	94.33

RA 2.2: Salient Methods on StackOverflow

Methods present in text of posts are known as *salient* methods. For a method to be salient, it must be central to a code example or have some discussion defining its purpose or describing its usage [49]. Since Android has the highest number of salient free-form text code elements [49], we investigate whether salient methods in Android StackOverflow help discovering relationships between API methods.

Our training set consists of all clusters of API methods present in natural language text of StackOverflow answers posts exclusively. It consists of a total of 29,114 transactions and 47,021 items. The test set is the one previously used by other developed alternatives, it involves 2,520 transactions and 2,521 items. We experimentally determined our minimum support and confidence thresholds; our setting consists of a minimum support count of 5 and a minimum confidence of 70 percent. Table 10 reports the findings obtained with the salient methods based-approach which we can interpret as follows.

Leveraging salient methods in API documentation we find that we can predict changes in individual Apps with an average precision of 62 percent and an average recall of 17 percent. Compared to our baseline, we increase our precision by 27 points, but decrease our recall by 5 points.

The high increase in precision is expected since salient API methods are relevant for problems described in StackOverflow posts [49]. In effect, unlike contextual API elements that are necessary details when solving a problem, salient methods are central to code examples. Additionally, focusing on highly-voted answers posts makes our prediction even more accurate.

The slight difference obtained in terms of recall is likely due to the fact that some contextual API methods from code snippets – that may be relevant to a problem/task at hand – have not been exploited since only API methods mentioned in natural language descriptions of StackOverflow are leveraged.

We conclude that salient methods can help discovering relationships between API methods.

2.4.5 RA 3: Combing the Best Approaches

To reveal the extent to which combining development history and documentation can enhance the prediction of related API methods, we suggest a model based on the integration of best predictive models from source code change history and documentation, i.e. similar Apps and salient methods-based Approaches.

Table 11 summaries the findings obtained with this approach and which can be

Table 11: Results of Change Predictability using Source Code Change History and Informal API Documentation: $P_{M'}$ = Precision , $R_{M'}$ = Recall, $F_{M'}$ = F-measure.

StackOverflow Results					
Metrics	*1Q*	*Median*	*Mean*	*3Q*	*Max*
$P_{M'}$	43.65	67.38	**73.17**	94.56	100
$R_{M'}$	7.13	18.20	**27.08**	29.65	100
$F_{M'}$	8.90	19.24	**31.65**	47.18	84.56

interpreted as follows.

Combining development history and informal API documentation we find that we can predict changes in individual Apps with an average precision of 73 percent and an average recall of 27 percent. Compared to our baseline, we increase our precision by 38 points and recall by 5 points.

The improvement brought on precision is expected since we are combining rules from best change history and API documentation predictive models. It clearly confirms that API documentation, specifically the StackOverflow discussions reflect Android API methods changes, which corroborates the findings by recent research works [33].

The slight increase in recall, even though not significant from a statistical point of view, reveals that API documentation can be leveraged to complement approaches that exploit source code change history when predicting relationships between API calls.

Overall, we conclude that combining development history and informal API documentation can help increase the predictive power of models aiming at discovering relationships between API calls.

2.5 Threats to validity

We have analyzed a large set of open-source Apps, i.e. 230 Android Apps belonging to twenty-two different category. Our study involves a total number of 64,429 transactions and 51,908 items from source code while it analyzes 36,440 transactions and 50,185 items from API documentation in total. Although the studied applications belong to different domains, we cannot claim that their version histories and corresponding API documentation would be representative for all kinds of software projects.

Transactions created from version histories do not specify the order of the individual atomic changes because these changes are committed simultaneously under Git which does not record such information. Similarly, transactions built by clustering API methods discussed in StackOverflow posts do not keep track of the notion of order of API methods since a transaction in such a case is identified by the date and time of the document unit (i.e. post) mentioning a set of specific API methods. Hence, our evaluation does not take into account the actual order of changed/discussed API methods.

Right now we have mined the API changes rules using all transactions regardless of their relevance/quality. One could investigate rules generated from recent development history as they may reflect more the current state of a software application. In our study, this threat is mitigated by the fact that, in general, Android Apps are recent (long history is of maximum 6-7 years).

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 StackOverflow because, often, it contains discussions triggered by professional developers about API elements that perplex them. However, other sources of information can be leveraged as well. Examples of such sources include bug reports repositories, developers' emails, code reviews, chats, etc.

We identified API elements discussed in StackOverflow posts using an accurate approach, ACE [49]. Yet, we cannot guarantee that similar results will be obtained with different resources and–or using other approaches for linking API elements with documentation.

We have partially analyzed the source code corresponding to the Git commits of each single application to extract API elements. Partially analyzing source code of version histories is not an easy problem. It has been shown that it is an undecidable problem [12]. Therefore, it may be that the PPA approach used in this work has failed to identify some API calls for example. However, we are confident, given its extensive evaluation on a large sample of open-source projects that it only produces 2.7 percent of erroneous types when analyzing a single class without its dependencies while it is able to correctly recover on average 91.2 percent [12].

We evaluated the predictive power of our suggested approach using predictive models based on the notion of training and test datasets to generate related API methods. Another way of evaluating our investigation and its impact could be by means of user studies with professional developers who can use our tool within their Integrated Development Environment when navigating through the source code to understand the relationships between API methods calls that are part of their software change tasks.

2.6 Related Work

We first focus on relevant contributions to the prediction of source code changes by mining version histories (Section 2.6.1). We then present state-of-the-art approaches that mined documentation to discover API usage patterns (Section 2.6.2).

2.6.1 Mining Version Histories to Predict Source Code Changes

Zimmermann *et al.* suggested an approach called ROSE that uses association rule mining on CVS data to recommend source code that is potentially relevant to a given fragment of source code [73]. Similarly to Zimmermann *et al.*, we apply data mining techniques, specifically, the widely used association mining algorithm,i.e. Apriori. However, our approach expands on ROSE by predicting changes developers make to API method calls instead of predicting changes associations between files or methods. In addition, Zimmermann *et al.* exploit version history of single projects while we suggest the use of development history across a community of (all or similar) applications to predict changes in API methods. Furthermore, we leverage informal API documentation (i.e. StackOverflow) to predict dependencies between APIs methods. Finally, we propose an enhanced predictive model using both both development history and StackOverflow to predict changes in APIs methods.

Independently from Zimmermann *et al.*, Annie Ying developed an approach that also uses association rule mining, namely frequent pattern mining, on CVS version archives to predict file change patterns [?]. She showed the usefulness of her approach on the Eclipse and Mozilla open-source projects by evaluating the predictability and interestingness of the recommendations produced for actual modification tasks on these systems. In contrast to our approach, Ying's tool exploits source code changes of single projects only. In addition, it is limited to changes between files, not changes made at the level of finer-grained entities or API methods calls such as the case in this work.

Bruce *et al.* [8] developed a system for IDEs that can learn from existing code repository and made relevance suggestion to developers about changes. Bruce *et al.* [9] also proposed a concept of IDE that can help developers based on information retrieve from other developers work.

Xing and Stroulia [68] have attempted to detect class-co-evolution by mining versions of UML diagrams. This method relies on the UMLDiff algorithm that, given a sequence of UML class models of a system, surfaces the design-level changes over its life span. Their findings are promising in terms of facilitating the overall understanding of system evolution and the planning of future maintenance activities. However, their approach has not yet been empirically evaluated on a large scale.

Hassan and Holt predicted change propagation for fine-grained entities by investigating a set of heuristics that leverage historical change and static dependencies. Their research addresses the following question: *How does a change in one source code entity propagate to other entities?* They empirically validated their results using data obtained by analyzing the development history for five large open-source software systems [22]. Differently from this work, our approach relies on the use of mining association rules.

Sayyad-Shirabad *et al.* used inductive learning to determine concepts of pertinence between logically coupled files [57, 58, 56]. They presented the notion of *Relevance Relation* to represent relations among entities in a software system and showed how classification learning can be used to model relevance relations.

Michail applied data mining on the source code of programming libraries to detect reuse patterns in form of association [39] or generalized association rules [40]. The entities (items) of these rules consist of method invocation, inheritance, instantiation, or overriding. These prior works do not bring any empirical evidence on the quality of the discovered patterns. Differently from this work, our approach mine source code changes across a community of applications instead of leveraging change history of individual projects.

Kagdi *et al.* have suggested a set of heuristics for grouping change-sets found in source code repositories. Their approach provides a sequence of changed-files along with a partial temporal ordering information. The technique has been evaluated on a subset of KDE source-code repository [26]. In contrast with our approach which focuses on analyzing the co-evolution of API elements, this work mainly investigates association files rules. Rysselberghe *et al.* mined frequently applied changes in a version control system and suggested a two-dimensions visualization technique to help recognize change-relevant information [55].

Gall *et al.* have developed an approach that exploits information in a release history such as the version number of programs, modules and subsystems, as well as change reports to discover logical dependencies and change patterns among modules [18].

Mockus *et al.* [42] have introduced a quantitative method to assess the extent of the coordination problem among sites by identifying tightly coupled work items that are recorded in software change management systems or chunks, that span several sites. This work defines a process for determining chunks that are candidates to be moved to different developments sites.

Shirabad *et al.* [59] suggested the application of inductive methods to data extracted from both the source code and software maintenance records. Their approach indicates which files, in a legacy system, are relevant to each other in the context of software maintenance.

Uddin *et al.* [64] have developed a technique that analyzes the evolution of an API's integration in client programs. Their technique identifies temporal API usage patterns, i.e. a sequence of usage pattern that are implemented in distinct development phases to detect significant changes in API usage. The initial development of such a technique can help learn more about an API usage and inform both API developers and consumers.

Recently, researchers [66] have studied how the fault- and change-proneness of APIs used by Android Apps relates to applications' lack of success, estimated from user ratings. The findings obtained by means of a large empirical evaluation have shown that APIs used by successful Apps are significantly less fault- and change-prone than APIs used by unsuccessful Apps including when changes affected method signatures and especially public methods. Instead, changes to the set of exceptions thrown by methods did not significantly relate with the App success. In contrast to this work in which the authors have mined the APIs entire change history from the APIs Git repositories to analyze the relationship between heavy changes/bugs introduced in APIs and the success of Android app, we mine APIs methods calls changes at the level of each commit from the Git repositories of the studied Apps. In addition, we use partial programming analysis to address the challenging problem of identifying APIs methods calls in partial programs corresponding to commits made by developers to source code of Android Apps. Finally, our purpose is not to discover the factors that impact the success of Android Apps but to suggest APIs methods relevant to a software developer task.

2.6.2 Leveraging Documentation to Study API Usage Patterns

Zhong *et al* have developed an API usage mining framework and its supporting tool called MAPO (Mining API usage Pattern from Open source repository) to automatically mine API usage patterns from open-source repositories and recommend the mined patterns and their associated code snippets upon a programmer's requests [72]. They used frequent subsequence mining with clustering to mine API usage patterns from code snippets. Oppositely to this work, our approach uses both development history and documentation to discover relationships between API methods.

Textual similarity of log messages [69] or program code [4] have been exploited to guide developers during their engineering activities. Further improvements on such works have been suggested by HIPIKAT [11] that uses other sources (other than source code) such as mail archives and online documentation. Differently from our approach, these tools discover dependencies between files or classes rather than between fine-grained entities such as API methods. Additionally, they emphasize on high recall instead of high precision as in our investigation.

Buse *et al* have suggested an automatic technique for mining and synthesizing human-readable documentation of programs interfaces [10]. Their approach takes into account a combination of path data flow analysis, clustering, and pattern abstraction. It produces results in the form of well-typed code snippets which document initialization, methods calls, assignments, looping constructs, and exception handling.

The authors evaluated their approach by means of a user study involving over 150 participants. Their findings show that 82 percent of the generated examples were judged as good as gold-standard human written documentation and 94 percent were preferred over the state-of-the-art code search.

Ponzanelli *et al.* [47] proposed an approach, that given a context in the Eclipse IDE, suggest relevant discussions from StackOverflow that relate to a code snippet under analysis. The evaluation of this approach during maintenance and development tasks shows that it significantly help developers in completing the experimental tasks. Additionally, the participants agree on the usefulness of its features and usability. Differently from this work, we mine highly rated answers in Stackoverflow to suggest API usage patterns. Other researchers such as Bacchelli *et al.* have developed an Eclipse plug-in namely, Seahawk, that assist programmers using StackOverflow [47]. Seahawk formulates queries automatically from an active context in the Eclipse IDE, displays a ranked list of results, and links discussions and source code fragments by means of language-independent annotations.

A recent research work [33] has investigated how changes occurring to Android APIs trigger questions and activity in StackOverfow. In this investigation, the authors have used as a proxy of the developer community, the questions posted in SO and tagged to Android-related labels; and as a proxy of the changes, they analyzed developers' commits for the analysis of methods changes. In general, the findings have shown that developers in the SO community react to changes in Android APIs. They provide important insights about the use of social media to learn about changes in software ecosystems, as well as about the importance of developing recommender systems that leverage documentation such as StackOverflow to recommend changes related to API elements, in particular methods, that are part of a software developer task.

Other researchers [44] have analyzed developers' collaboration mined from different sources of information including mailing lists, issue trackers, and IRC chat log as well as their co-change activities captured from versioning systems. The results of this study have shown that social network metrics captured from mailing lists and issue tracker reflect well the developers' activity, while this is not the case for chats. Motivated by the fact that social media and bug reports reflect developers' change activity and by the fact that software ecosystems changes triggers developers' discussions in StackOverflow, we predict changes in API methods calls using not only development history but also API documentation, in particular, StackOverflow.

2.7 Conclusion and Learned Lessons

In this study, we mined source code change history and informal API documentation to help a developer identify high-level relevant changes for both internal and external API methods. We have empirically validated our hypothesis that the suggested approaches can provide valuable recommendations by applying them to 230 open-source different Android Apps. Results of an extensive empirical evaluation have shown that our techniques, in particular, the similar Apps and salient methods approaches are able to accurately provide change suggestions of API methods with a high precision: 60-70 percent. Additionally, combining best predictive models from development history and StackOverflow has been proven to have almost the same performance as the similar Apps-based approach. We thus bring empirical evidence on the fact that leveraging source code change history across multiple similar Apps helps enhance the API methods change predictions made for individual Apps. In addition, we show that informal API documentation reflects code change activities by developers which corroborates the findings by recent works [33, 44]. More importantly, such an informal API documentation helps discovering pertinent dependencies between API methods with high accuracy when salient methods are leveraged. We believe 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 analyses-based techniques.

What lessons *we* have learned from this investigation, and what are our suggestions? In the following, we present our plans for future work:

- **Defect-Prone APIs.** APIs evolve quickly. Their change and fault proneness have been proven to impact the success of the Apps using them [66]. To help developers identify defect-prone API methods, one could identify changes that induce bugs by linking transactions to bug databases using appropriate algorithms such as SZZ [27] to determine whether particular API methods tend to change whenever a bug occurs. Transactions that consist of buggy changes should be given priority as their entities are defect-prone. Therefore, our approach can be used not only to to help developers understand and cope with the fast evolution of APIs but also to contribute to the enhancement of software quality, it can also be applied in the context of refactoring activities where developers need an awareness about the change and defect-prone API elements to be tackled first.

- **Other Types of Documentation.** Other yet unexploited sources of information are bug reports, developers' emails, chats, and code reviews which contains important information about code elements [49, 5]. We used StackOverflow since it has large set (22 millions) of posts discussing Android API elements. Further data sources can be leveraged using appropriate approaches such as the ACE [49] technique used in this study or Baker specifically designed for extracting code examples from API documentation.

- **Application in Practical Settings.** The evaluation of our suggested approaches was performed using predictive models based on the notion of training and test datasets. We plan to evaluate our approach in the context of concrete tasks, e.g. modification tasks or bug-fixing activities where developers have to understand and discover the evolution of APIs methods pertinent to their tasks. The user study will involve professional developers in industrial settings who are interested in evaluating the practical interestingness of our approach. In this way, we would be able to show its impact for both researchers and practitioners interested in using a tool that helps them understanding the evolution of internal and external APIs used by software projects. The approach will be used in the form, for example, of an Eclipse plug-in. The tool will have a client server architecture where the server will take into account all the treatments concerning the grouping of atomic changes and clustering of API methods on StackOverflow, their conversion into transactions, as well as the mining of API methods change suggestions. In the client part, a user can enter the name of an API method that is part of his task and the tool will provide him with the list of pertinent rules.

- **Selecting Task-Pertinent History**. The suggested approach searches for patterns in the learning history which consists of all transactions from both past and recent history. Patterns inferred from history evolve during time; some patterns that were relevant become of less importance or not valid later in a software project. The reason is that older development history reflects older software projects' programming habits and strategies that may not be followed by developers anymore. In the case of APIs, this becomes even more challenging since they evolve fast. One research question to be addressed could be as follows:

  Would exploiting recent learning history improve the predictability of our approach and thus the quality of the APIs suggestions provided to developers?

  For such a purpose, we will consider only recent transactions (e.g. from last year) or implement an implicit aging for rules by assigning higher rates to new transactions than older ones for example.

Chapter 3

Software metrics to suggest potential license violations.

On average 2371 Apps are added everyday on Google play store [29], such statistics make developers aware of the very high speed of App development and updates. In this competitive market, developers may look for ways to speed up development to remain ahead of competitors. In this race for developing their Apps at higher speed, developers may be tempted to copy code from open-source Apps having GPL license. This may lead to legal issues. If a software has a GPL license then each user must be allowed to use, share and modify the original source code. Any change or combination of this source code with other code must be released to public [19].

The license of an open-source App may be unsuited for a proprietary App and if some copied code fragment is found in proprietary App then it can put the owner of proprietary App into legal troubles.

In our investigation we have used 150 open-source Apps from F-Droid[1] and 950 proprietary Apps from Google play store[2]. Based on the idea of Bertillonage similar to Davies et al [14] we developed a tool and provided a set of measures that can be used for finding similarity between proprietary Apps and open-source Apps. However Davies work was limited to finding the provenience of a software entity present in a software, we find the violation made by proprietary Apps by considering the license of open-source Apps. Our similarity measures are based on Android API calls an App made.

Since the source code of proprietary Apps is not available, we have extracted Android API calls from the binaries of open-source as well as proprietary source by using jclassinfo tool that we will discuss in more detail methodology. Our goal is to use this information to determine if closed source Apps are copying code from open source Apps. Our similarity measures narrow down the search effort for finding license violations.

The objectives of this research work are as follows:

1. Create a set of similarity measures based on the API calls an Apps makes
2. Create a "universe" of Android application with their API calls parsed
3. Report the proportion of Apps that have potential violations.
4. Release the tool and allow developers to download APK files to check if violations had been made against their project.

[1]https://f-droid.org/
[2]https://play.google.com/store?hl=en

3.1 Research Approach

3.1.1 The Concept of Bertillonage

With the invention of photo Camera in mid 19th century, police departments of various European countries started using it to identify criminals. They simply used name, age and photographs of criminals to identify them. Soon criminals realized that by giving false informations such as name and age they can hide their identity and it became a burden for police department to look large number of photographs to identify a criminal. To overcome this problem Bertillon proposed a scientific method based on anthropological technique to reduce the searching task for identifying a criminal. He suggested that if a criminal record consist of name, age and some physical measurement such as height, length of right ear, length of left foot etc then photos can be organized based on those biometrics data and it will reduce the set of photographs need to examine for a given suspect. In the honour of its inventor this method termed as Bertillonage and used for two decades as one of the best method for identification [25, 41].

Android App Bertillonage Metrics: Similar to software Bertillonage metrics introduced by Davis et al [14], we introduce our own Android App Bertillionage metrics. These metrics help in reducing the search space to find license violation among Android Apps. Our metrics are mainly based on the number of Android APIs shared between different Android Apps. We used different measures to find the best possible metrics to detect the copying.

API count based (baseline): Android Apps use Android APIs to implement functionality. We examine all API call made by Apps and find out how many Android API method calls are being shared by both the Apps, The larger percent of APIs being shared the greater the likelihood of copied code.

High overlap Apps: There are many small Android Apps which calls only few APIs and shows high percentage of its APIs being similar to other Apps. To get rid of those Apps we considered Apps that shares at least 50 distinct Android API method calls. The larger the number of APIs sharing among Apps, the more similar an App might be with other.

Category based: Apps that implement similar functionality (e.g., two restaurant Apps) are likely to have more in common. We want to understand if there is partial copying of code between Apps in the same category. The App category is defined on Google Play store.

Set based: We counted the number of methods per class and class per package in each App and cluster the similar outcomes in the same category. We use these categories to find similarities between Apps. Our assumption was that Apps in same category will have more similarity than Apps in different categories.

Size based: We also used release size information of each App to calculate similarity.

3.2 Motivational Example

During 2011, at AnDevCon (Android development conference) [38] in San Francisco a company called OpenLogic that advices companies on the proper use of open source project ran a test to find out license violation among Apps and they came up with non trivial number of license violation among different Apps.

They used 635 Apps in their study, 523 of them were from Apple Store and 112 from Google Play Store. Their findings confirms that 71% of Apps using open source licenses were not compliant. OpenLogic sells a tool called OSS Deep Discovery. This tool examines source code and binary to identify open source code and its license based on the dependencies of the software on other libraries and systems. When bundled together, there can be combinations of licenses that violate legal agreements. In contrast, to examine software dependencies, we use the calls to the Android libraries to indicate possible copying of code. They look for linking violation, we look for evidence of copied code.

3.3 Data and Methodology

We collected two types of Apps, open source from F-Droid [3] and proprietary Apps from Google Play store [4]. From F-Droid we collected 150 Apps out which 141 Apps were having GPL license and from Google play store we collected 950 Apps from different categories. We randomly picked 950 Proprietary Apps and downloaded

[3]https://f-droid.org/

[4]https://play.google.com/store?hl=en

their APK using Android Market API [5] and parse them using our own scripts. From F-Droid we downloaded 150 APK using our own crawler.

3.3.1 Phase 1 - Identifying API calls used by Apps

To identify API elements used by the Apps, we downloaded the Android PacKage (APK) files for each release of each App. An APK file is a package file format used for the distribution and installation of application software and middle-ware onto operating systems such as Google's Android OS. These files contain information such as the resources and software code including the complied classes in the DEX (Dalvik EXecutable) format. Then, we extracted the API elements used by each App from the downloaded APK files following the three main phases.

- For each App release, we converted the APK file to jars files using the dex2jar[6] API;
- For each jar file corresponding to each App release, we use the JClassInfo[7] tool to extract API elements, i.e. packages, classes, and methods used by the Apps;
- For each App release, we pruned the code elements obtained from JClassInfo and kept only the ones belonging to the **Android.*** package.

We wrote appropriate scripts to extract API calls and elements using the JclassInfo tool. Jclassinfo is written in C. It reads Java class files and provides information

[5]https://code.google.com/p/android-market-api/
[6]*http://code.google.com/p/dex2jar*
[7]*http://jclassinfo.sourceforge.net*

about referenced packages, classes/interfaces as well as methods, etc. We extracted non-Android API elements, however, we were unable to use these in the analysis because the names of the classes and methods had been obfuscated. The obfuscated code elements could not be compared between releases to determine which had been added or removed.

3.3.2 Phase 2 - Collecting App's informations

Table 12: License of few open source App used in Study

Open-source	License
CurrentWidget: Battery Monitor	GPLv3+
CSipSimple	GPLv3+
Bankdroid	GPLv3+
DroidFish Chess	GPLv3+
DieDroid	GPLv3+
Tipitaka	GPLv3+
DieDroid	GPLv3+
Bankdroid	GPLv3+
AnagramSolver	GPLv3+
AnkiDroid Flashcards	GPLv3+
AnyMemo: Flash Card Study	GPLv3+
DroidBeard	GPLv3+
Bankdroid	GPLv3+
ElloShare	GPLv3+
Der Bund ePaper Downloader	GPLv3+
ePUBator	GPLv3+
AnkiDroid Flashcards	GPLv3+
DeskCon	GPLv3+
Call Meter 3G: THE monitor app	GPLv3+
Andor's Trail	GPLv3+
DSub	GPLv3+
aLogcat (free) - logcat	GPLv3+
Tipitaka	GPLv3+
Document Viewer	GPLv3+
DroidZebra Reversi	GPLv3+

Table 13: Information about few Studied Apps.

Title_app	version	Category	size(KB)
ProArchery	6.1	Sports	2118
Newspapers from Mexico	1	News & Magazines	1102
My Traffic	1.2.25	Productivity	877
Traffic Master Lite	2.0.7	Casual	818
AS	2.0.005	Sports	3488
Retro Clock Widget	2.1.5	Personalization	1661
Magic Trick	1	Casual	608
SCANNER PRO - QR Code Reader	2.6	Tools	2843
Revolver	2	Entertainment	2422
TouchRetouch Free	3.2.2	Photography	2823
Bad Blood TCG	1.0.15	Card	1017
Training Memory - Game	1.2	Puzzle	789
Cool 3D Gallery	1.007	Media & Video	8488
Zen Table Tennis Lite	2.0.5	Sports	1319
Yoo Ninja! Free	1.13	Arcade	1070
eCalc	1.03	Lifestyle	1059
Bubble Shoot Royal Deluxe	1.2.7	Casual	2833
Funny Warp	3.2	Entertainment	1052
SafetyGPS V3	3.0.2	Social	3954
Talking Tom Cat Free	2.5	Entertainment	2441
B.Med Chat	1.0.5	Finance	1224
Era Architects, Mumbai - India	0.21.13220.34478	Business	2613
Counter desert strike	2.3	Arcade	2226
Kira Kira Jewel(No.9)Free	1.0.0	Personalization	927
Korean in a Month Free	1.12	Education	2941

App id is common identifier for Apps on Google play store and F-Droid. We used it to get the information such as category of App, release size of App and its license. We randomly picked 950 App ids from Google play store using our script and used Android market API to get release size and category of each App for our research. Similarly we picked 150 random App ids from F-Droid and executed our script to get the category, release size and license of each open-source downloaded from it. Table 13 shows examples of the information gathered for Apps. Similarly Table 12 shows license information of various open source Apps.

3.3.3 Phase 3 - Calculating Similarities between Apps

We only considered the public Android API calls extracted from APKs for our study. In our study, we removed all the Android API calls that were present in more than 30% of Android Apps. This filters out most of the common APIs needed for general Android development. We calculated the similarity between two Apps based on the number of public Android API method calls they shared. For example,

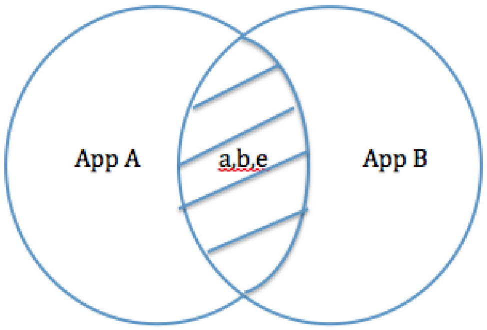


Figure 1: Pictorial representation of similarity

An App A has the following Android API method calls, $\left\{ \text{a,b,c,d,e,f,g,h,i,j} \right\}$

Similarly App B has the following Android API method calls, $\left\{ \text{x,y,z,a,b,e,m} \right\}$

From Figure 1 we see that both A and B shares the API methods a, b, and e. So, if we calculate similarity based on the API calls sharing we will get the following. Suppose Q_A is the Android API call made by Android App A, Q_B is the Android API call made by Android App B.

Similarity of A with B $= Q_A \cap Q_B / \ Q_A$

Similarity of B with A $= Q_A \cap Q_B / \ Q_B$

How similar A is with B = 3/10.

How similar B is with A = 3/7.

3.3.4 Phase 4 - Android App Bertillonage metrics

To narrow down the search space for finding license violation among our App corpus, we develop four metrics based on Android API calls sharing, method calls per class, App categories define on Google play store and released size of App.

High Overlap : While calculating similarities based on phase 3, we also keep track of the number of APIs being shared between both the Apps. If both the Apps shares at least 50 distinct Android API calls then we look further to find license violation. Our assumption was the more an App shares API calls with other, the greater the chances code has been copied.

Category : We use Android market place API and our script to get the category of each Android App defined on Google play store. We calculated the similarity between Apps falling in the same category. Apps who behave in similar fashion and have similar functionality are likely to share more APIs and there may be high chances of violating license.

Set Based: Instead of calculating similarities of an App with all the other available Apps, we decided to calculate similarities of the given App with only those Apps that have same number of API method calls per class. We assume that if an App copies some code fragments from other source then it might not change the Android API calls in it, although there might be some addition or deletion of code, the method per class ratio might be close to the original source. We categorize API method calls per class into categories like 0, 1, 2 ... . If an App makes less than 1 API method calls per class then we categories it to 0. Similarly if an App make less than 1 API method calls per class we categorized it to 1 and so on.

Released size: We collected the released size information of each Android App in KB and to narrow down our search operation to find the similarity between each App we come up with one more assumption that if an App is very similar to an other App, then it might be possible that their release size is somehow similar. So, instead of finding similarity of an App with all the other Apps, we only examine those which are having similar released size of that given App. We also categorized release size into different categories with difference of 500 KB.

3.4 API call Copy Example

This section illustrates examples of common APIs found in different Apps. We have hidden the name of Apps in examples as we cannot reveal them because of legal issues.

From Table 14, Using **High Overlap** approach we found that proprietary App "Y1" shares 506 distinct Android API calls with open-source App "X1". It shows that "Y1" is 53.04% similar with "X1", and Similarly "X1" is 24.89 % similar to "Y1".

Table 15 shows different Android APIs which are being shared between "X"

Table 14: Similarities between Few Apps

opensource	google_title	code_sharing	opensource_similarities	google_similarity
X1	Y1	506	0.248893261	0.530398323
X2	Y2	178	0.098396904	0.5129683
X3	Y3	1429	0.78819636	0.953302201
X4	Y4	796	0.46252179	0.623335944
X5	Y5	761	0.442184776	0.532913165
X6	Y6	449	0.309015829	0.470649895
X7	Y7	1249	0.365204678	0.532849829
X8	Y8	1427	0.787093216	0.900315457
X9	Y9	156	0.092089728	0.641975309
X10	Y10	1015	0.304896365	0.524006195
X11	Y11	994	0.306979617	0.553144129
X12	Y12	612	0.338308458	0.584527221
X13	Y13	615	0.302508608	0.618712274
X14	Y14	223	0.188823031	0.256027555
X15	Y15	747	0.524947294	0.584964761

and "Y". These are not common Android APIs, as we have already removed those Android APIs which are present in 30% of Apps. This shows that there might be some copy code.

Table 15: Android API calls Sharing between Apps

OS	Android_API	P_App	Android_API
X	android.view.accessibility.AccessibilityNodeInfo.setScrollable	Y	android.view.accessibility.AccessibilityNodeInfo.setScrollable
X	android.support.v4.view.PagerTitleStripIcs.setSingleLineAllCaps	Y	android.support.v4.view.PagerTitleStripIcs.setSingleLineAllCaps
X	android.view.View$AccessibilityDelegate.getAccessibilityNodeProvider	Y	android.view.View$AccessibilityDelegate.getAccessibilityNodeProvider
X	android.support.v4.app.FragmentManagerImpl.performPendingDeferredStart	Y	android.support.v4.app.FragmentManagerImpl.performPendingDeferredStart
X	android.support.v4.app.FragmentManagerImpl.dispatchPrepareOptionsMenu	Y	android.support.v4.app.FragmentManagerImpl.dispatchPrepareOptionsMenu
X	android.support.v4.view.ViewCompat.setImportantForAccessibility	Y	android.support.v4.view.ViewCompat.setImportantForAccessibility
X	android.view.SoundEffectConstants.getContantForFocusDirection	Y	android.view.SoundEffectConstants.getContantForFocusDirection
X	android.support.v4.view.AccessibilityDelegateCompat.sendAccessibilityEvent	Y	android.support.v4.view.AccessibilityDelegateCompat.sendAccessibilityEvent
X	android.support.v4.view.ViewCompat$ViewCompatImpl.canScrollHorizontally	Y	android.support.v4.view.ViewCompat$ViewCompatImpl.canScrollHorizontally
X	android.support.v4.view.ViewCompatICS.setAccessibilityDelegate	Y	android.support.v4.view.ViewCompatICS.setAccessibilityDelegate

3.5 Empirical Evaluation

We used box plots to support our findings. A boxplot gives a pictorial representation of group data through their quantile. We plotted the box plot of proprietary Apps with open-source Apps based on number of Android API calls sharing. With the help of box plot we show that our metrics help us to narrow down the search window for finding license violations.

API calls count : We calculated similarities between each open-source App and proprietary App based on the number of API methods they share. In Figure 2, we can see that the overlap of Apps is low with a median of 10%. Most Apps do not have much in common and are not copies of each other, which is what we would expect. However, there are a number of outliers that do have a lot in common.

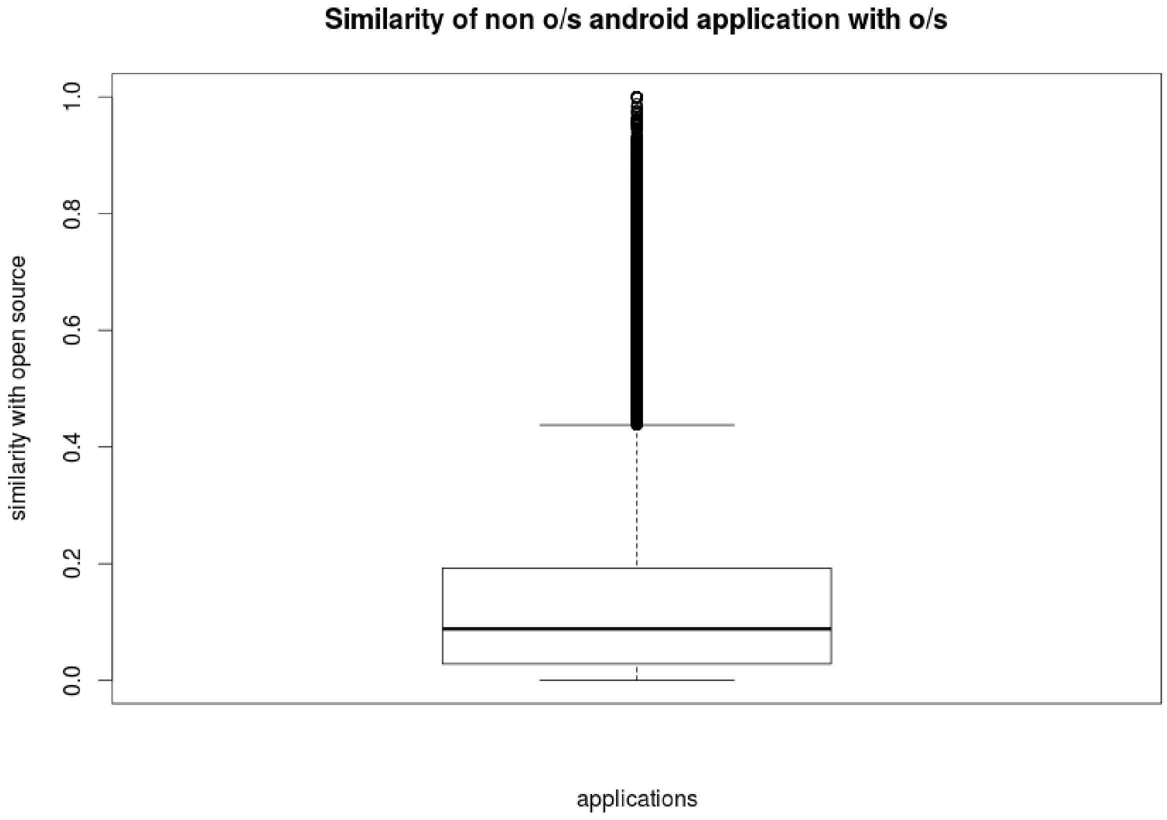


Figure 2: Similarity of Proprietary App with Open-source App

High Overlap : The above box plot contains lots of outlier that have many Apps in common. To examine these outliers, we only consider the Apps that have 50 distinct API method calls in common. As expected the median similarity rises to an overlap of 40%. All the below explained metrics are based on only the high overlap Apps. We calculated the similarity using below metrics only if the Apps shares at least 50 distinct API method calls.

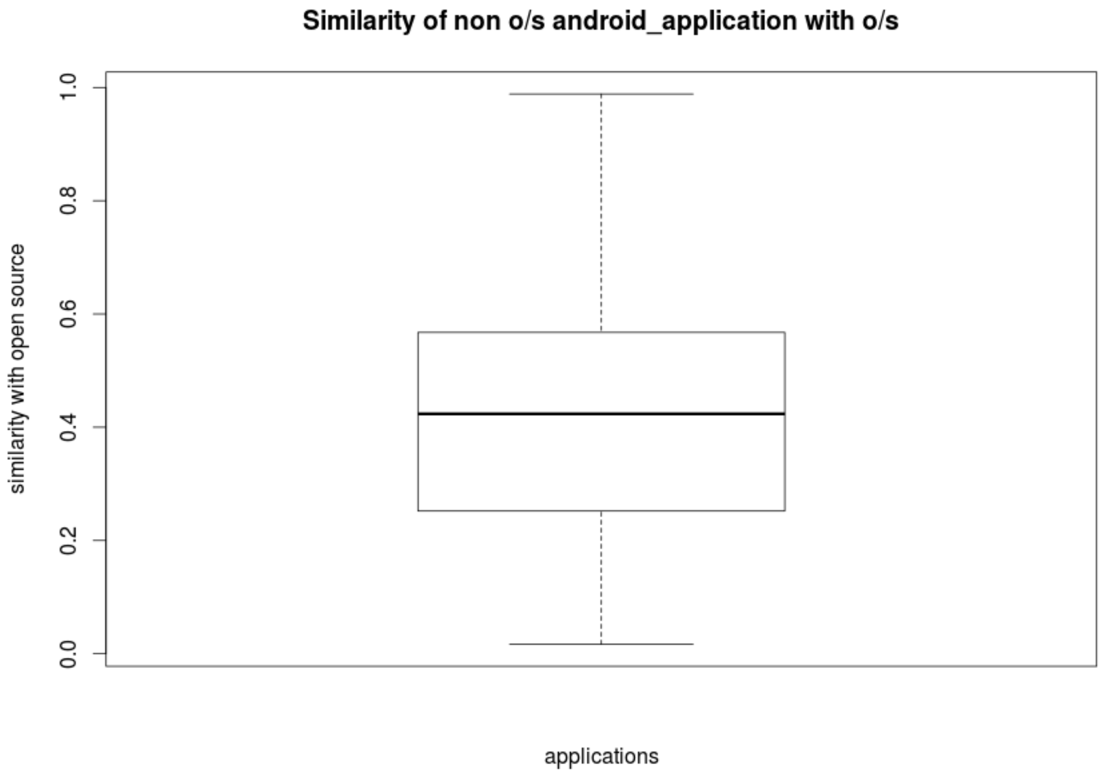


Figure 3: Similarity of Proprietary App with Open-source App, high Overlap

Similar Category: We examined Apps that implement similar functionality and expect that they will have more in common. We also conjecture that Apps that fall in the same category might be copying more code from each other. The plot below shows that Apps in the same category have a high median similarity of 50%, which is much higher than the median similarity of 10% when comparing all Apps with high overlap.

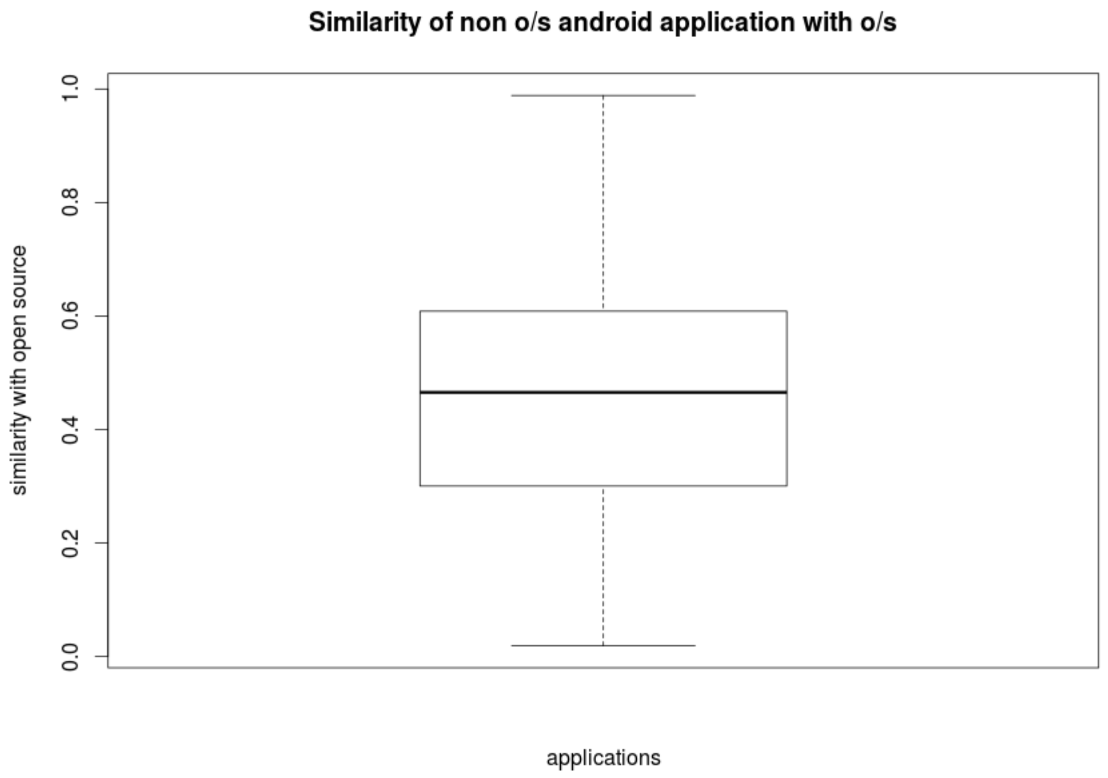


Figure 4: Similarity of Proprietary App with Open-source App having same category

API calls per Class : We assume that if an App copies some code fragments from other source then it might not change the Android API calls in it, although there might be some addition or deletion of code, the API calls per class will be similar to the original source.

In Figure 5 Application 1 are those proprietary applications which are having similar number of API calls per class with open-source Apps whereas application 2 are those proprietary applications that are having different number of API calls per class. From Figure 5 we can easily find that similar the number of API calls per class between two APPs, more the changes of being copied.

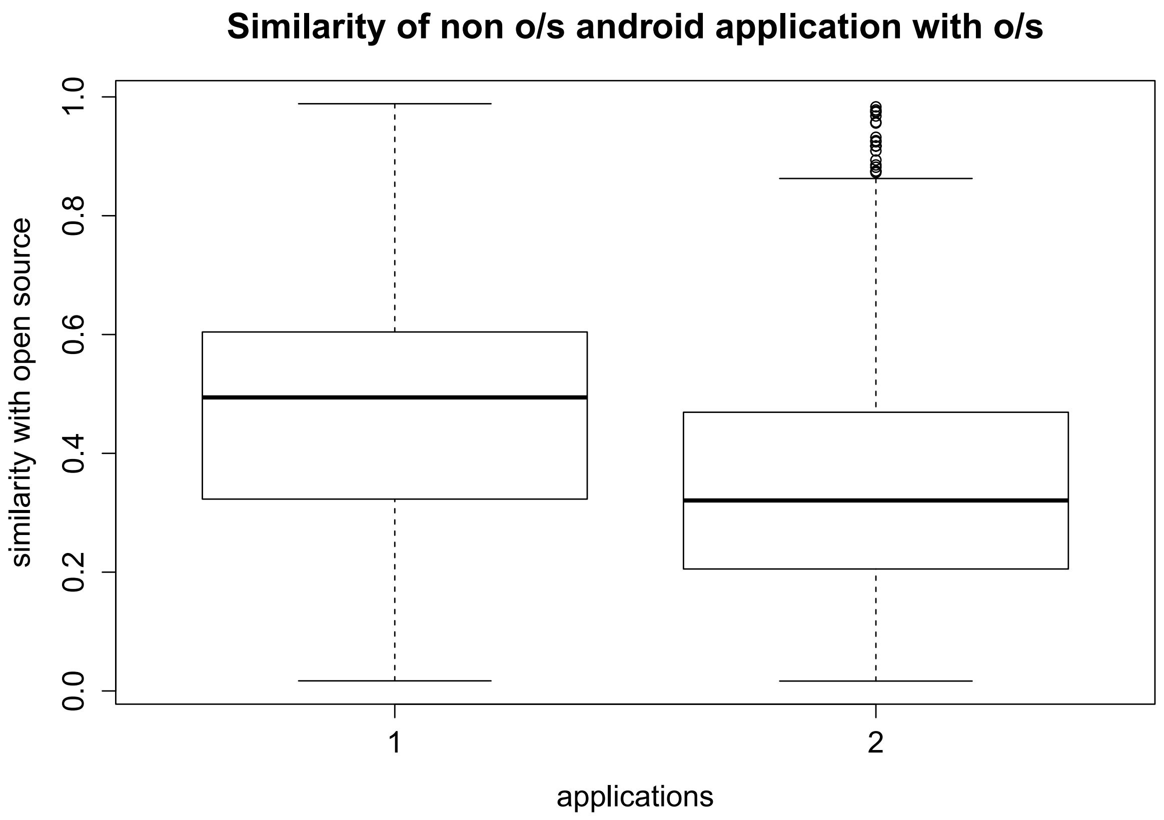


Figure 5: Similarity of Proprietary App with Open-source App having similar number of APIs call per class

Released size : App having same functionality and copied from some open-source App, may have same released size. Based on this fact we plotted the box plot of similarity between open-source with proprietary App.

The above conjecture is not supported by the box plot in Figure 6. As their might be lots of Apps whose release size might be equal to the release size of other Apps which are totally different.

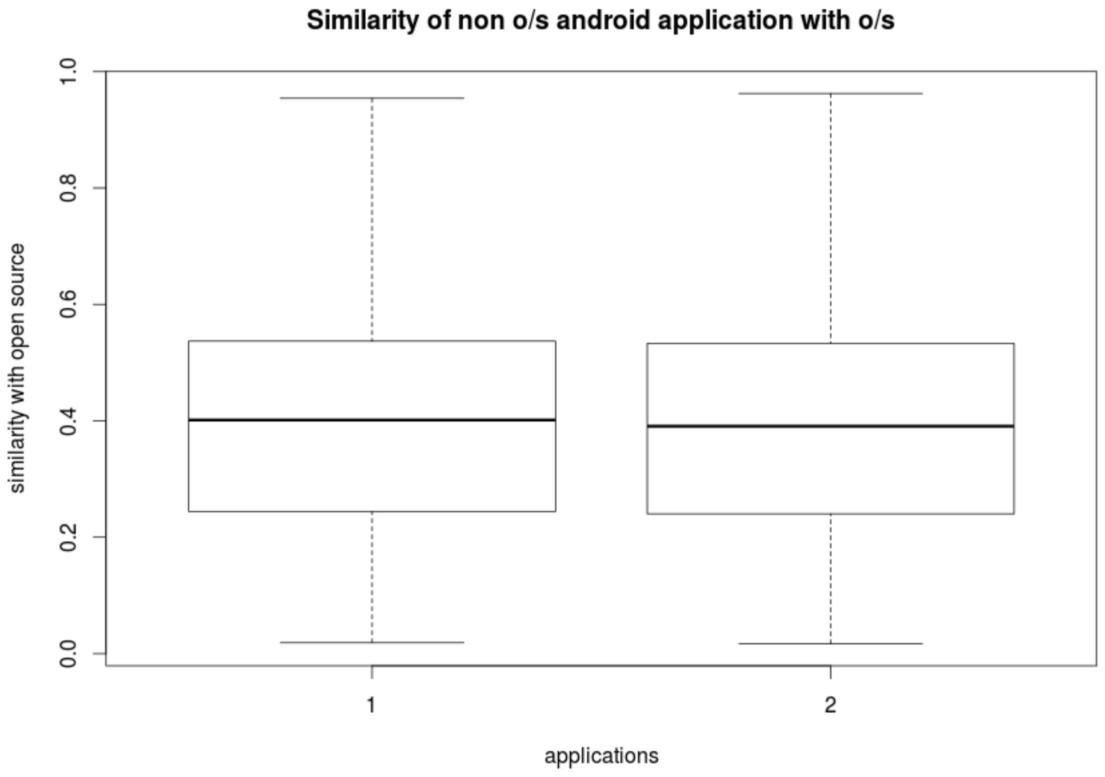


Figure 6: Similarity of Proprietary App with Open-source App having similar released size

3.6 Manual Inspection

To support our findings from boxplot, we would have liked to contact some Android open source App developers and interviewed them. Since our research involves reverse engineering of proprietary Android Apps we first sought permission from the University. We contacted Me Bech, Associate legal counsel, Concordia University regarding this issue and his response is in the following quotation :

"I've been thinking about your concern. I think that it may be ok from a patent law perspective to reverse engineer the apps for the sole purpose of research. That being said, it may not be allowed from a contract law perspective; if you agree when

downloading (or in the terms of use) not to reverse engineer the app, that would be a contractual undertaking separate from patent law. I would be surprised if most apps did not bar reverse engineering in their terms of use. In other words, it appears to me that this intended behaviour is risky. It appears to me to be a significant risk (and beyond the purview of "for research purposes only") if you reverse engineer and then make a point of specifically notifying those whose copyright may be infringed of this fact"

Manual inspection remains the only way to support our findings. So we did our own manual analysis of 10 applications with the highest similarity scores (based on percentage of API calls being shared by proprietary Apps with open source Apps). As we can not reveal the name of Apps so, we used term "X" to denote the open source and term "Y" to denote the proprietary Apps. We installed 10 Apps from Google play store in our mobile phone and analyzed their functionality and appearance to find whether Apps are copying the code from others or it is a part of normal Android development.

As we did not interview the developers, we present a table to measure the extent of copying. Our table is based on our manual inspection of Android Apps based on their GUIs and working style. This table contains two columns, the first column defines the severity of chances of copying and it ranges from high to negligible and points associated with each severity. The other column categorizes all the instances of copying into different severity measures.

After installing Apps on mobile, we started looking for the examples from Table

Table 16: Manual similarity measures

Severity	Points	Example of Copying
High	5	Same error message, Same text in welcome screen, Same behavior at *cornercases*^{*8}, Same structure and flow of dialog boxes Same behavior and text formating of check boxes Particular hanged case
Moderate	3	Same appearance of App's icon, Same directory structure and layout of App, Same settings preferences, Same layout of buttons, Same layout of background images, Accessing same user information
Low	1	Same functionality, Same touch response, Size of buttons, Same dependency.
Negligible	0	Background color, Text color, Font size of text

16. For each example we found, we took the particular point associated with it and sum it up. The higher the points, higher are the chances of copying code from open source. We found out several examples of copying instances between an open source Apps and proprietary Apps.

We installed Y1 and X1 on our phone. App Y1 is used to organize file content whereas X1 is a banking App. The first thing that can be noticed is the appearance of icons in both the App. The second noticeable thing is the way pop up windows appear in both of them and the way check-box work and appears. The total points for the above said pair adds up-to 11.

Similarly we downloaded X2 an open-source App and Y2 a proprietary App. Both the Apps belongs to the same category on Google play store. The layout of both the Apps are very similar, accessing the same user information i.e phone number, photo gallery, same font size of texts, same behavior at corner cases, same appearance of App's icon. If we sum up the points associated with each example, it comes out to be 13.

Table 17: Result of manual inspection

Open source	Proprietary	Total Points	Example of Copying
X1	Y1	11	Same appearance of App's icon, Same directory structure and layout of App, Having same color for check-boxes
X2	Y2	13	Layout of both the Apps are very much similar, Accessing same user information, Same behavior at corner cases, Same font size of texts, Same appearance of App's icon
X3	Y3	8	Same appearance of App's icon, Background Color Same structure and flow of dialog boxes
X4	Y4	17	Same behavior and text formating of check boxes, Particular hanged case, Same dependency Same layout of background images, Accessing same user information,
X5	Y5	4	Same text colors, Almost same delay in touch response, Same directory structure of App.
X6	Y6	0	Same background color, Same functionality (No points added as both belongs to same Category)
X7	Y7	9	Same structure and flow of dialog boxes, Same settings preferences, Size of buttons.
X8	Y8	6	Same appearance and size of App's icon Same directory structure and layout of App
X9	Y9	10	Same layout of buttons, Accessing same user information, Same layout of background images, Same delay in touch response
X10	Y10	2	Size of buttons, Same functionality.

We present a table containing 10 pairs of Apps and represent total points associated with that pairs based on manual similarity measures. These points can be used to conclude whether an App is copying other or not.

As these App pairs already share lots of API calls and show high percentage of similarity score based on API calls sharing so, we did manual inspection on them to support our findings. The result from Table 17 shows pairs having higher points reflecting some sign of copying. Pairs such as (X1, Y1), (X2, Y2), (X4,Y4) and (X9,Y9) shows high points on manual inspection as well as our similarity metrics reflect the same, so it is highly probable that proprietary App is copying the open source.

3.7 Threats to Validity

This section discuss the main threats to our research. We analyzed a large set of Android Apps that consists of 950 Proprietary Apps and 150 open-source Apps. Although the set is larger and belongs to different category of Android Application but we cannot guarantee that it will represent all kind of Android Apps.

We downloaded APK of Apps and applied reverse engineering on it to get the APIs. Lots of optimization takes place while converting .class file to APK such as - multiple classes are included in a single DEX file, same constant used in multiple class files are included only once in DEX output to conserve space. These DEX files are again modified when installed into the mobile device, such as addition of new libraries, swapping of byte order in certain data etc. All this process adds new elements and modifies the original source code so APIs calls extracted from APKs may be slightly different from the API calls made by original source code of that App.

We used dex2jar and jclassinfo for reverse engineering which performs very well when we compared with the actual API calls made by original source code but still cannot confirm that extracted APIs will remain the same if different tools and approach will be used.

To support our findings, we did manual inspections that may differ from person to person as everyone have different perspective to look at the same things. So, different people can report different findings for the same App while inspecting Apps, even we might have missed some of the important cases that can be a clear case of code copy example.

3.8 Discussion

There are clearly some Apps that are very similar in the calls they make to the Android API calls. This conclusion remains when we remove the most common API methods across the community of Apps. The main difficulty is in differentiating copying from the similarity that would naturally exist between Apps that implement similar functions.

One promising technique that we investigated includes clustering API calls based on the App's class that contains them. For example, if code was copied, then in both systems the API calls should come from the same class within the Apps, even if the class name has been changed. However, if the Apps were simply similar, then one would expect a different set of classes making the same calls. The other techniques such as grouping Apps based on categories, high overlap also helped us in reducing the search effort for finding copying.

We conducted manual inspection of 10 Apps. Although the majority of pairs shows some sign of similarity on manual inspection but there were few pairs of Apps that were showing high percentage of similarity based on Android APIs sharing but shows no sign of code copying on manual inspection. This exhibits that there might be some Android APIs that remains there even after removing most common APIs during our study. So, we need to have some other measures as well to explore code copying and ultimately finding license violations.

One of the most important and promising technique can be interviewing Android developers. This can lead to better result and better understanding of similarity metrics proposed by us.

3.9 Related Work

Clone detection remains one of the most hot topic in software engineering research and lots of researchers have studied about the source of copied code present in a software [54, 6, 31, 7]. These previous works was mainly concentrated around finding the origin and evaluation of clones. Later on, such research shifted to clone maintenance and genealogy [28, 63].

All these research provided a basic foundation for our research i.e., to find the proprietary Apps that are copying code from open-source and violating its GPL license. Similar to Davies et al [14] we also find out the provenance across multiple applications from their binary but our work is based on API calls made by the App whereas their work was based on matching code elements from their binaries. Again their work was limited to Java-based software systems that are easy to decompile from binary to class files but we worked on the APKs of Android Apps that are really challenging and time consuming to decompile. Our main concern was to find the license violation among Android Apps whereas their main concern was to find provenance and its source.

In the past, Di Penta et al. [45] inferred the license of a class file present in jar archive using Google code search. They used Google code search to get the information about the origin of included class files and there license. They extracted the licenses of included classes to inform the developers which classes they can combine or use with their system. Our approach was different from them as we extracted class file from bytecode and applied reverse engineering to get the Android method calls from it. We used these calls to identify how similar two applications are.

Similar to Davis et al. [14], we proposed our own software metrics to find similarity between two Android Apps.

3.10 Conclusion and Future Work

Keeping in mind that there are 1.3 millions Android Apps available on Google play store [48], determining the license violation of open-source Apps by proprietary Apps become a very much expensive and challenging job. In this research we studied the Android API calls of different Apps and found out similarity between them. Given the two APks of two distinct Android Apps, our approach can found the similarity between them.

We introduced new similarity measures, to reduce the search effort to find the violation and empirically validated our hypothesis that these measures can be helpful in finding the license violations among Apps. We demonstrated the effectiveness of this approach by simple box-plots of different similarity measures and manual inspection of few Android Apps. We found that even manual inspection were supporting our results and showing high sign of copying among those Apps which were already being pointed out by our approach.

The similarity measures provided by us is only a beginning and it can be expanded more to accurately determine the copied App with reduced effort. In future, we may send the list of potential violations to the OSS App developers, and interview them to see if they found the information useful. We also want to create a site that allow developers to submit in their APK and see if it was similar to other Apps, unfortunately, we would not be able to tell them which ones because of legal issues. The horizon of similarity measures needs to accommodate different parameters so that it can also be applied on large software systems.

3.11 Tool

Based on our research work we developed a tool for Ubuntu that can be used for calculating similarity between two or more Android Apps.

Our tool uses the same approach discussed by us in methodology section. We have provided the detail about its installation and dependency in Appendix A.

Chapter 4

Conclusion

The Android App market is growing at a tremendous pace that can be understood by looking at the number of Android Apps present on Google Play store. There are more than 1.3 million Android Apps present on Google play store. This vast data opens doors for lots of un-addressed research questions. We utilized this data to address some research question on the development and copyright violation of Apps.

Our research work can be divided into two major parts. In the first part we developed a model that can be used to predict next likely to be added or removed API calls in an App by mining its version history and on-line informal documentation. The second part was to developed a method which we later turned into a tool that can be used to reduce the search efforts to find out which proprietary App is copying code from open-source Apps and ultimately violating its license.

4.1 Guiding App developer about APIs changes

We believe 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. Of course, the more there is to learn from (development and documentation) history especially recent one, the more and better predictions can be made:

- Leveraging source code changes across multiple similar applications helps predict changes in internal and external API methods with a high precision in 72 percent of cases. This confirms the following statement: *the more there is to learn from history, the more and better change suggestions can be made.*

- Documentation can also be used to understand the co-evolution of API elements. StackOverflow was able to accurately show co-evolving API methods in about 65 percent of cases. Additionally, documentation increases the recall and thus documentation can complement development history towards a better prediction of co-evolving API elements and in particular methods that are parts of a software developer task.

- The predictive power of our approach increases when exploiting recent source code change or documentation history. Therefore, approaches and tool that aim to guide developers when performing code change tasks should focus on investigating recent (change or development) learning history rather than old one.

The evaluation of our approach was done using predictive models created on training sets and evaluated on test sets. We plan to evaluate our approach in the context of concrete tasks, e.g. modification tasks or bug-fixing activities where developers have to understand and discover the evolution of APIs elements pertinent to their tasks. The user study will involve professional developers in industrial settings who are interested in evaluating the interestingness of our approach. In this way, we would be able to show its impact for both researchers and practitioners interested in using a tool that helps them understanding the evolution of internal and external APIs used by software projects.

4.2 Software metrics to suggest potential license violations

Our main contribution in this research was to develop a tool that can find the similar Android API calls made by different Android applications. On the basis of Android APIs calls, we developed some software metrics that can reduce the effort of finding similarity between different Apps.

We proposed four kinds of software metrics based on API sharing and we empirically as well as manually evaluated them and found them efficient. We can conclude the following findings from our research:

- Apps that are in the same categories(Based on Google play store) are having more Android APIs in common then the Apps from different categories. Keeping in mind we have already removed the most common APIs, this makes us suspicious about these Apps.

- Apps that make similar number of Android APIs calls per class shows lots of common Android APIs being shared by those Apps. This technique can also work where simple code matching technique fails i.e. in the case where the developer changes the names of copied classes.
- Manual inspection shows that there are few proprietary Apps that might be copying code from open-source Apps.

Although our tool can be used for reducing the search effort, it is only the beginning. This tool can be very useful for open-source developers to save their Apps from being exploited by others. In the future, other parameters can also be included in this tool to make it more reliable and accurate. We want developers to download and use it as well as give us feedback so that we can improve it more in the future.

Bibliography

[1] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software Practice and Experience*, 23:589–616, 1993.

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

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

[4] David L. Atkins. Version Sensitive Editing: Change History as a Programming Tool. In Boris Magnusson, editor, *SCM*, volume 1439 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 1998. ISBN 3-540-64733-3.

[5] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts.

In *32nd ACM/IEEE International Conference on Software Engineering*, pages 375–384, 2010.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8779-7. URL `http://dl.acm.org/citation.cfm?id=850947.853341`.

[7] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE*, 33(9): 577–591, 2007.

[8] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from Examples to Improve Code Completion Systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595728. URL `http://doi.acm.org/10.1145/1595696.1595728`.

[9] Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini. IDE 2.0: Collective Intelligence in Software Development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 53–58, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0427-6. doi: 10.

1145/1882362.1882374. URL `http://doi.acm.org/10.1145/1882362.1882374`.

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

[11] Davor Cubranić and Gail C. Murph. Hipikat: recommending pertinent software development artifacts. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, 2003. IEEE Computer Society Press.

[12] Barthelemy Dagenais and Laurie J. Hendren. Enabling static analysis for partial Java programs. In Gail E. Harris, editor, *OOPSLA*, pages 313–328. ACM, 2008. ISBN 978-1-60558-215-3.

[13] Barthelemy Dagenais and Martin P. Robillard. Recovering traceability links between an API and its learning resources. In Martin Glinz, Gail C. Murphy, and Mauro Pezze, editors, *ICSE*, pages 47–57. IEEE, 2012. ISBN 978-1-4673-1067-3. URL `http://dblp.uni-trier.de/db/conf/icse/icse2012.html#DagenaisR12`.

[14] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. Software Bertillonage: Finding the Provenance of an Entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 183–192, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.

1145/1985441.1985468. URL http://doi.acm.org/10.1145/1985441.1985468.

[15] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 19:992–1030, 1998.

[16] Chris Eleftheriadis. State of the Mobile Developer Mindshare, July 2013. URL http://mashable.com/2013/07/24/google-play-1-million/.

[17] Etienne M Gagnon, Laurie J Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *Static Analysis*, pages 199–219. Springer, 2000.

[18] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8779-7. URL http://dl.acm.org/citation.cfm?id=850947.853338.

[19] GNU. GNU General Public License, June 2007. URL https://www.gnu.org/copyleft/gpl.html.

[20] Carl Gould, Zhendong Su, and Premkumar T. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *ICSE*, pages 645–654.

IEEE Computer Society, 2004. ISBN 0-7695-2163-0. URL `http://dblp.uni-trier.de/db/conf/icse/icse2004.html#GouldSD04`.

[21] Rajiv Gupta and Mary Lou Soffa. A framework for partial data flow analysis. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 4–13. IEEE, 1994.

[22] Ahmed E. Hassan and Richard C. Holt. Predicting Change Propagation in Software Systems. In *ICSM*, pages 284–293. IEEE Computer Society, 2004. ISBN 0-7695-2213-0.

[23] Ahmed E. Hassan and Tao Xie. Mining Software Engineering Data. In *Proc. 34th International Conference on Software Engineering (ICSE 2012), Tutorial*, June 2012. URL `http://www.cs.illinois.edu/homes/taoxie/publications.htm`.

[24] Sture Holm. A Simple Sequentially Rejective Bonferroni Test Procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979.

[25] J.Siegal, P.Saukko, and G.Knupfer. *Encyclopedia of Forensic Science*. 2000.

[26] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining sequences of changed-files from version histories. In Stephan Diehl, Harald Gall, and Ahmed E. Hassan, editors, *MSR*, pages 47–53. ACM, 2006. ISBN 1-59593-397-2. URL `http://dblp.uni-trier.de/db/conf/msr/msr2006.html#KagdiYM06`.

[27] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Software Eng.*, 39(6):757–773, 2013.

[28] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An Empirical Study of Code Clone Genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, September 2005. ISSN 0163-5948. doi: 10.1145/1095430.1081737. URL `http://doi.acm.org/10.1145/1095430.1081737`.

[29] John Koetsier. 700K of the 1.2M apps available for iPhone, Android, and Windows are zombies, August 2013. URL `http://venturebeat.com/2013/08/26/700k-of-the-1-2m-apps-available-for-iphone-android-and-windows-are-zombies/`.

[30] Rainer Koppler. A Systematic Approach to Fuzzy Parsing. *Softw., Pract. Exper.*, 27(6):637–649, 1997. URL `http://dblp.uni-trier.de/db/journals/spe/spe27.html#Koppler97`.

[31] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1. doi: 10.1109/WCRE.2006.18. URL `http://dx.doi.org/10.1109/WCRE.2006.18`.

[32] l. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking App behavior against app descriptions. In *ESEC/SIGSOFT FSE'13*, pages 1025–1035, 2014.

[33] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 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 2014, pages 83–94, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2879-1. doi: 10.1145/2597008.2597155. URL `http://doi.acm.org/10.1145/2597008.2597155`.

[34] Ingrid Lunden. Canalys Q2: 68 % Of All Smartphones Shipped Were Android; ChinaâĂŹs The Biggest Market By A Wide Margin, Aug 2012. URL `http://techcrunch.com/2012/08/02/canalys-q2-68-of-all-smartphones-shipped-were-android-chinas-the-biggest-market-by-a-wide-margin/`.

[35] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest Q&A site in the west. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 2857–2866. ACM, 2011. ISBN 978-1-4503-0228-9. URL `http://doi.acm.org/10.1145/1978942.1979366`.

[36] A. Marcus, J. I. Maletic, and A. Sergeyev. Recovery of Traceability Links Between Software Documentation and Source Code. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):811–836, 2005.

[37] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay Spinuzzi. Building More Usable APIs. *IEEE Software*, 15(3):78–86, 1998.

[38] Scott Merrill. Potential Open Source License Violations In Android and iOS Apps?, March 2011. URL `http://techcrunch.com/2011/03/08/potential-open-source-license-violations-in-android-and-ios-apps/`.

[39] Amir Michail. Data Mining Library Reuse Patterns in User-Selected Applications. In *ASE*, pages 24–33, 1999. URL `http://dblp.uni-trier.de/db/conf/kbse/ase1999.html#Michail99`.

[40] Amir Michail. Data mining library reuse patterns using generalized association rules. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *ICSE*, pages 167–176. ACM, 2000. ISBN 1-58113-206-9. URL `http://dblp.uni-trier.de/db/conf/icse/icse2000.html#Michail00`.

[41] M.M.Houck and J.A.Siegal. *Fundamentals of Forensic Science*. 2006.

[42] Audris Mockus and David M. Weiss. Globalization by Chunking: A Quantitative Approach. *IEEE Software*, 18(2):30–37, 2001.

[43] Leon Moonen. Generating Robust Parsers Using Island Grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*,

WCRE '01, pages 13–22, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1303-4. URL `http://dl.acm.org/citation.cfm?id=832308.837160`.

[44] Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. How DevelopersâĂŹ Collaborations Identified from Different Sources Tell us About Code Changes. In *ICSM*, pages 251–260, 2014.

[45] Massimiliano Di Penta, Daniel M. GermÃąn, and Giuliano Antoniol. Identifying licensing of jar archives using a code-search approach. In *MSR'10*, pages 151–160, 2010.

[46] Kavita Philip, Medha Umarji, Megha Agarwala, Susan Elliott Sim, Rosalva Gallardo-Valencia, Cristina V. Lopes, and Sukanya Ratanotayanon. Software Reuse Through Methodical Component Reuse and Amethodical Snippet Remixing. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1361–1370, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1086-4. doi: 10.1145/2145204.2145407. URL `http://doi.acm.org/10.1145/2145204.2145407`.

[47] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: stack overflow in the IDE. In *ICSE*, pages 1295–1298, 2013.

[48] The Statistics Portal. Number of apps available in leading app stores as of July 2014, July 2014. URL `http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/`.

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

[50] Martin Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16:703–732, 2011. ISSN 1382-3256. URL `http://dx.doi.org/10.1007/s10664-010-9150-8`.

[51] Martin P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, 2009.

[52] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment Class Analysis for Testing of Polymorphism in Java Software. *IEEE Transactions on Software Engineering*, 30(6):372–387.

[53] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *Proceedings of the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '99, pages 235–252. ACM, 1999.

[54] Chanchal Kumar Roy and James R. Cordy. A Survey on Software Clone Detection Research. *SCHOOL OF COMPUTING TR 2007-541, QUEENS' UNIVERSITY*, 115, 2007.

[55] Filip Van Rysselberghe and Serge Demeyer. Mining Version Control Systems for FACs (Frequently Applied Changes). pages 48–52, 2004.

[56] J. Sayyad-Shirabad, T.C. Lethbridge, and S. Matwin. Mining the Software Change Repository of a Legacy Telephony System. In *International Workshop on Mining Software Repositories (MSR 2004)*, pages 53–57. IEEE Computer Society, 2004. ISBN 0-7695-1905-9.

[57] Jelber Sayyad-Shirabad, Timothy C. Lethbridge, and Stan Matwin. Supporting maintenance of legacy software with data mining techniques. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON*, page 11. IBM, 2000.

[58] Jelber Sayyad-Shirabad, Timothy Lethbridge, and Stan Matwin. Mining the Maintenance History of a Legacy Software System. In *ICSM*, pages 95–104. IEEE Computer Society, 2003. ISBN 0-7695-1905-9.

[59] J. Shirabad, Timothy Lethbridge, and Stan Matwin. Supporting Software Maintenance by Mining Software Update Records. In *ICSM*, pages 22–31, 2001. URL `http://dblp.uni-trier.de/db/conf/icsm/icsm2001.html#ShirabadLM01`.

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

[61] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM. ISBN

978-1-4503-2756-5. doi: 10.1145/2568225.2568313. URL http://doi.acm.org/10.1145/2568225.2568313.

[62] Suresh Thummalapenta and Tao Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321663. URL http://doi.acm.org/10.1145/1321631.1321663.

[63] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[64] Gias Uddin, Barthelemy Dagenais, and Martin P. Robillard. Analyzing temporal API usage patterns. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 456–459. IEEE, 2011. ISBN 978-1-4577-1638-6. URL http://dblp.uni-trier.de/db/conf/kbse/ase2011.html#UddinDR11.

[65] C.J. van Rijsbergen. *Information Retrieval*. 1979.

[66] Mario Linares Vasquez, Gabriele Bavota, Carlos Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API change and fault proneness: a threat to the success of Android apps. In *ESEC/SIGSOFT FSE'13*, pages 477–487, 2013.

[67] Christina Warren. Google Play Hits 1 Million Apps, July 2013. URL `http://mashable.com/2013/07/24/google-play-1-million/`.

[68] Zhenchang Xing and Eleni Stroulia. Data-mining in Support of Detecting Class Co-evolution. In Frank Maurer and GÃijnther Ruhe, editors, *SEKE*, pages 123–128, 2004. ISBN 1-891706-14-4.

[69] Andrew Y Yao. Cvssearch: Searching through source code using cvs comments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 364. IEEE Computer Society, 2001.

[70] Annie T. T. Ying and Martin P. Robillard. Code fragment summarization. In *ESEC/SIGSOFT Foundations on Software Engineering*, pages 655–658. ACM, 2013. ISBN 978-1-4503-2237-9.

[71] Annie T. T. Ying and Martin P. Robillard. Selection and presentation practices for code example summarization. In *ACM SIGSOFT Foundations on Software Engineering*, pages 460–471, 2014.

[72] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 318–343, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. URL `http://dx.doi.org/10.1007/978-3-642-03013-0_15`.

[73] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller.

Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

Appendix A

SimilarityFinder User Manual

This tool can calculate similarity between two or more android Apps provide their APKs are given. It requires dependencies such as postgreSQL, Jclassinfo and dex2jar.

A.1 Installing Dependencies

You need to have administrative privilege to install these dependency:

1. Install jclassinfo
   (a) $ sudo apt-get install jclassinfo
2. Install PostgresSql
   (a) $ sudo apt-get install postgresql postgresql-contrib
   (b) create a user and database [Details for creating user and database is provided in next section]

3. Download Jex2jar

    (a) https://code.google.com/p/dex2jar/downloads/list

A.1.1 Creating Database and database user in PostgresSql

To do any operation in postgresSql, you need to create a "role", which provides authentication to database. To create a "role" and database follow the following steps:

1. After installing PostgresSql, type the command in terminal to create "role":

    (a) $ sudo -i -u postgres

    (b) $ createuser

        i. It will ask you following details:

        ii. $ Enter name of role to add: 'NAMEOFUSER'

        iii. $ Shall the new role be a superuser? (y/n) y

        iv. $ \ q

        v. $ exit

2. To create database, follow the following command:

    (a) $ su – 'NAMEOFUSER'

    (b) $ createdb 'NAME OF DATABASE'

A.2 Installation

Download the tool package and use Ubuntu software center to install.

1. Double click on downloaded tool package

2. It will open in Ubuntu software center and there you can see install button.

3. Click the install button.

4. You might get warning please ignore it.

If you do not want to use Ubuntu software center, then extract the ∗.deb file and you can find an executable "toolsimilarity" in that which can be used for finding similarity.

A.3 Usage

Once you installed it or extracted the executable file, you can use terminal to run the tool. You need to provide two configuration file while running this tool. The first configuration file should contain the name of database and downloaded path of dex2jar. The other file should contain the location of APKs need to be studied. you can use configuration file from text but need to edit it with required information.

To run the tool :

1. If you have installed it on your machine

   (a) $ toolsimilarity sam.conf sam.txt

2. If you want to use executable

   (a) $ ·\toolsimilarity sam.conf sam.txt

A.4 Output

This tool will produce two table in the database provided by user. The first table "similar_api" contains all the Android APIs common to both the APPs and the other table "Similarities" contains the information about percentage of API similarity.

A.4.1 Disclaimer

We accepts no liability for the output of this tool. Any output result presented by this tool is solely for research purpose. We do not guarantee that it works 100 percent. WARNING: We accepts no liability for any damage caused by this tool.