API Recommendation for Event-Driven Android Application Development

Weizhao Yuan[a,b], Hoang H. Nguyen[b], Lingxiao Jiang[b,*], Yuting Chen[a,*], Jianjun Zhao[c,**], Haibo Yu[d,**]

[a]*Department of Computer Science and Engineering, Shanghai Jiao Tong University, P.R. China*
[b]*School of Information Systems, Singapore Management University, Singapore*
[c]*Faculty of Information Science and Electrical Engineering, Kyushu University, Japan*
[d]*School of Software, Shanghai Jiao Tong University, P.R. China*

Abstract

Context: Software development is increasingly dependent on existing libraries. Developers need help to find suitable library APIs. Although many studies have been proposed to recommend relevant functional APIs that can be invoked for implementing a functionality, few studies have paid attention to an orthogonal need associated with event-driven programming frameworks, such as the Android framework. In addition to invoking functional APIs, Android developers need to know where to place functional code according to various events that may be triggered within the framework.
Objective: This paper aims to develop an API recommendation engine for Android application development that can recommend both (1) functional APIs for implementing a functionality and (2) the event callback APIs that are to be overridden to contain the functional code.
Method: We carry out an empirical study on actual Android programming questions from StackOverflow to confirm the need of recommending callbacks. Then we build Android-specific API databases to contain the correlations among various functionalities and APIs, based on customized parsing of code

*Principle corresponding author
**Corresponding author
Email addresses: weizhaoy@163.com (Weizhao Yuan), mr.erichoang@gmail.com (Hoang H. Nguyen), lxjiang@smu.edu.sg (Lingxiao Jiang), chenyt@cs.sjtu.edu.cn (Yuting Chen), zhao@ait.kyushu-u.ac.jp (Jianjun Zhao), haibo_yu@sjtu.edu.cn (Haibo Yu)

snippets and natural language processing of texts in Android tutorials and SDK documents, and then textual and code similarity metrics are adapted for recommending relevant APIs.
Results: We have evaluated our prototype recommendation engine, named LibraryGuru, with about 1,500 questions on Android programming from StackOverflow, and demonstrated that our top-5 results on recommending callbacks and functional APIs can on estimate achieve up to 43.5% and 50.9% respectively in precision, 24.6% and 32.5% respectively in mean average precision (MAP) scores, and 51.1% and 44.0% respectively in recall.
Conclusion: We conclude that it is important and possible to recommend both functional APIs and callbacks for Android application development, and future work is needed to take more data sources into consideration to make more relevant recommendations for developers' needs.

Keywords: Android programming, event callbacks, code search, API recommendation, information retrieval

---

1. Introduction

Software developments nowadays often need to utilize existing libraries. Developers, especially those who are new to a development environment, often need help in learning the capabilities of the environment and getting familiar with APIs provided in the environment for them to implement functionalities according to requirements. A prominent example is for Android application development where developers often need to comply with the specific framework settings and utilize specific APIs in Android to implement functionalities for their applications.

Android application developers need to know not only the APIs that can be invoked for implementing various functionalities, which we call *functional APIs*, but also the event callback APIs that are defined and managed by Android and can be overridden by developers to customize and extend default behaviours of the Android framework, which we call *event callbacks* or simply *callbacks*. Functional APIs tell developers *what* to do for implementing a functionality, while callbacks tell developers *where* to place the implementation code. Both kinds of APIs are important for implementing a functionality properly, especially for an event-driven programming framework, such as Android, where control flows or call relations among the APIs are not explicitly presented to developers.

To illustrate the differences between functional and callback APIs, Figure 1 shows a code snippet in an Android application for creating an animation of a rocket launch. The main purpose of the code is to construct an Android window (by extending Android's **`Activity`** class). When the Android system executes the application, it implicitly invokes ***onCreate***, which is one of the Android framework's application life-cycle event callbacks that are invoked by the framework when the application is in various states. Application developers cannot control how a callback can be invoked, but they can define and control what to be executed inside the callback. In this example, the body of `onCreate` loads a rocket image and a sequence of snapshots of rocket movements (by invoking **findViewById**, **setBackgroundResource**, and **getBackground**), and when the rocket image is clicked by a user, it starts to play the sequence of snapshots to simulate the launch of the rocket (by invoking `rocketAnimation`.**start**). The developers need to invoke **setOnClickListener** to tell the framework to monitor click actions, but there is no explicit control flow in the code that links the click-event callback function ***onClick*** with other parts of the code. So, *callbacks* (e.g., onCreate and onClick) are in fact a kind of inversion of control, where the control (i.e., invocation of the functions) is in the hand of the Android framework, instead of the application code; the control of *functional APIs* is in the hand of the application code. A functional API can be invoked to provide a specific functionality, while a callback can be overridden to contain code of any functionality. Using *both callback and functional APIs* appropriately together are necessary for implementing well-behaving Android applications.

There have been many studies on recommending relevant libraries and APIs for developers to use when they are writing code according to certain feature requirements (e.g., [1–4]). Some studies also find or generate sample code to tell developers how to use (i.e., invoke) the APIs (e.g., [4–12]). However, as far as we know, those studies mostly focus on recommending functional APIs that can be *invoked* by developers. Few studies have paid attention to the orthogonal need of recommending callbacks that are available from the programming frameworks and can be *overridden* by developers. Such a need can be very common when building applications in event-driven programming environments, such as the Android framework. Without considering the implicit control flows and call relations among various event callback APIs, the recommended functional APIs and the generated sample API usages may not be fully understandable or reusable by developers.

Based on our empirical study of sampled questions related to Android

```java
public class RocketLaunch extends Activity {
  AnimationDrawable rocketAnimation;

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ImageView rocketImage = (ImageView) findViewById(R.id.rocket_image);
    rocketImage.setBackgroundResource(R.drawable.rocket_thrust_frames);
    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();

    rocketImage.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View view) {
        rocketAnimation.start();
      }
    });
...
```

Figure 1: Sample Functional APIs and Callbacks

development collected from StackOverflow (cf. Section 2), about 3% of the questions directly ask about situations that are dependent on callbacks, while overall about 35% of the questions and answers involve overriding callbacks, in addition to invoking functional APIs.

To fulfill both needs of functional APIs and callbacks for Android application development, this paper presents an API recommendation engine for Android that can recommend both (1) functional APIs for implementing feature requirements and (2) the callbacks that are suitable to be overridden to contain the functional code. Thus, we aim to address a common need for Android application developers: where I should implement this functionality.

A key component of our API recommendation engine relies on the discovery of descriptive contents for both functional APIs and callbacks. The actual techniques are based on Android-specific parsing of code snippets and natural language processing of textual descriptions contained in the public training tutorials and SDK documents available for Android developers[1]. Our engine recommends suitable APIs based on similarity measurements among the descriptions and developers' queries in the natural language, which is an idea similar to many other studies on information retrieval and code search.

We have implemented a prototype of the recommendation engine, named LibraryGuru[2]. We have evaluated the recommendation results of the prototype

[1] `https://developer.android.com/`

[2] Available at `http://libraryguru.info` for testing

on realistic Android development questions collected from StackOverflow. Even though many questions on StackOverflow have gotten answers, we do not use those answers to build the description databases, so that we can demonstrate the capability of our techniques for recommending APIs more clearly without the interfering benefits of historically correct answers for the questions. Our evaluation on about 1,500 questions shows that our prototype can recommend both functional APIs and callbacks with promising results: the top-5 results on recommending suitable callbacks and relevant functional APIs can achieve up to 43.5% and 50.9% in precision respectively, and 24.6% and 32.5% respectively measured in terms of mean average precision (MAP) scores. Although the data we collected and the tool we built in this paper were for Android application developments in Java, the idea of facilitating API recommendation by discovering correlations among functional and callback APIs and textual descriptions may be adapted for other event-driven programming frameworks, languages, libraries and APIs (e.g., building iOS apps in Objective-C and Swift, building front-end web apps with AngularJS in JavaScript, or programming Windows apps in Visual C++ and C#), and the tool may be used to recommend other libraries and APIs too if the processed API databases were replaced accordingly.

This paper extends our previous 2-page poster paper for ICSE 2018 [13], and its main contributions are as follows:

- We empirically identify the need of Android application developers for locating suitable callbacks to override during development.
- We build Android-specific API description databases indicating the correlations among various functionalities and APIs in Android, so that we enable the recommendations for both functional APIs and callbacks.
- We evaluate our tool with about 1,500 questions related to Android programming collected from StackOverflow, and show that our techniques for API recommendation, especially for callbacks, are promising.

The rest of the paper is organized as follows. Section 2 describes our empirical survey of the questions on StackOverflow and uses an example to motivate our study. Section 3 relates our study to much related work in the area of code search and recommendation. Section 4 presents our techniques in detail. Section 5 presents our evaluation results with respect to StackOverflow questions and discusses our limitations. Section 6 concludes with future work.

2. Motivation

This section presents a simple empirical study, based on questions and answers on StackOverflow, to illustrate how often Android application developers need help in functional and callback API recommendations.

2.1. The Needs

StackOverflow is a discussion platform where developers confer about programming skills and software engineering tasks. It allows its users to ask and answer questions. Accordingly, its voting mechanism is used to distinguish questions or answers that are good or bad. Users can up-vote the posts they like and down-vote the ones they dislike. In additional, tags are used to organize questions into different topics. Users need to attach at least one tag and up to five tags per question. For example, Figure 2 shows votes and tags for a sample question. This particular question is tagged with "android", "android-activity", "parent-child", and "pass-data"; it is likely a commonly appearing question as it is up-voted by 122 users and starred by 48 users (as shown by the numbers on the left side of the figure).

As of August 2017, StackOverflow exceeded 14 million questions [14]. One of the most discussed topics is "Android" with more than 1 million questions [15]. Among the Android questions, there are some typical sub-topics on "intent", "spinner", "activity", "database", "storage", "asynctask", "dialog", "connection", "location", etc..

StackOverflow provides APIs for retrieving questions and answers encoded in the JSON format. We retrieve about sixty questions per query from StackOverflow following the restrictions of the APIs, and repeat the queries many times. As the APIs may return many same questions for different queries, we remove duplicate ones through unique question IDs. Also, many questions do not have answers or up-votes; or, when there are answers for a question, no one up-votes any of the answers; or, the answers do not mention any Android API. We filter away such questions as their answers are not strong enough to be used as the ground truth to evaluate the results generated by our API recommendation engine. Finally, 1531 questions having the "Android" tag are collected.

We notice that about 35% (540) of the questions have one or more answer that contains names of callback events (e.g., `onClick`, `doInBackground` etc.). Among these questions, about 50 of them explicitly ask when or where to implement a functionality, which are related to event callbacks.

We realize that many developers, especially new ones, tend to ask high-level questions instead of describing specific issues. New Android programmers seldom have enough experience to understand the event-driven programming style in Android and the APIs related to their tasks, especially the event callback APIs. They are unlikely to ask event callbacks even when implementing their functionality requires them to override some callbacks, as implied by the ten times difference between the two sets of questions above (540 vs. 50). Therefore, we realize that recommending the necessary event callback APIs is one of the needs of developers.

2.2. Example

To further illustrate the needs faced by Android application developers, we show in Figure 2 a snapshot of the StackOverflow question No. 14292398:[3]

How to pass data from 2nd activity to 1st activity when pressed back?

122

48

I've 2 activities, **Activity1** and **Activity2**.

In `Activity1` I've a **Button** and **TextView**. When the button is clicked **Activity2** is started.

In `Activity2` I've an **EditText**.

I want to display the data retrieved from **EditText** in **Activity2** in the **TextView** in **Activity1** when back is pressed from **Activity2**.

can someone help me with the code to make this work?

android android-activity parent-child pass-data

share improve this question edited Feb 28 '17 at 7:05 asked Jan 12 '13 at 10:24

Figure 2: Sample Question: StackOverflow No. 14292398

"How to pass data from 2nd activity to 1st activity when pressed back?" This question title has two main parts: the first part asks about the functionality of passing data among activities; the second part asks when the functionality is needed, which is related to certain events in the Android framework. With the many APIs and event callbacks available, the Android framework provides developers with the flexibility to implement each of the two parts in various ways, and the answers to this question on StackOverflow indeed show this flexibility.

[3]`https://stackoverflow.com/questions/14292398/`

For the functionality of passing data around, a developer may use the `Intent` class and its API `putExtra` to store the data and send `Intent` objects among activities using the `startActivityForResult` API. Alternatively, the developer may use `SharedPreferences` and `Editor` classes to store the data in one activity, and use the `getSharedPreferences` API to retrieve data in another activity.

For the part on when/where the functionality is needed, a developer may override the `onBackPressed` callback to activate the data setting and passing, and override either the `onCreate`, `onResume`, or `onActivityResult` callbacks, depending on how the data is passed around, to retrieve the data.

We thus observe that the implementation of many functionalities in Android applications requires collaborative uses of both functional APIs and event callback APIs based on the even-driven framework, even though there are no explicit control flows or call relations among the methods. Recommending APIs for Android development thus needs to recommend both functional APIs for invocation and callbacks for overriding.

Our study in this paper aims to achieve this goal. Based on our prototype used in the evaluation, we are able to relate the functional APIs with suitable callbacks for various situations, making the API recommendation results more useful for Android application developers. Specifically for this question, our prototype recommends some relevant functional APIs and callbacks correctly. Table 1 shows the recommended functional APIs and callbacks and their combinations:

- The `Intent` class was ranked top in its recommendation results;
- The `startActivityForResult` API was among its top-10 results;
- The `onBackPressed` and the `onCreate` callbacks were the top-2 among recommended callbacks;
- The `onActivityResult` was among top-5;
- The `onResume` callback was among top-10; and
- The combined uses of `onCreate` and `Intent` and `startActivityForResult` were among top-10.

The results show that our prototype is promising in recommending both functional APIs and callbacks just based on the Android public tutorials and SDK documents, helping developers to program with an event-driven programming framework, without the need of the historical answers for the StackOverflow questions.

Rec. Rank	Relevant Functional APIs	Rec. Rank	Relevant Callbacks	Rec. Rank	Relevant Combinations
Top 1	`Intent`	Top 1	`onBackPressed`	Top 1	`onCreate + Intent`
Top 10	`startActivityForResult`	Top 2	`onCreate`	Top 10	`onCreate + startActivityForResult`
		Top 5	`onActivityResult`		
		Top 10	`onResume`		

Table 1: Recommendation Results (i.e., "Rec. Ranks") for the Sample Question

3. Related Work

Our work is related to numerous studies on code search and recommendation. Recommendation can happen at various stages during software development and maintenance processes and be useful for various tasks (e.g., requirement traceability, feature localization, developer allocation, library recommendation, API recommendation, code sample recommendation, bug triage, etc.), using a variety of data (e.g., source code, code change histories, documentations, discussion forums, bug databases, development contexts, etc.) [16]. Many of the techniques and tools can be applicable for Android development too, while our study is unique in the sense that we tailor the techniques for the needs of event-driven programming for Android applications in the context of the Android framework.

This section is by no means a comprehensive list of related work on code search and recommendation. It focuses on API recommendations for Android development, while explaining the relation between this study with others.

3.1. Code Recommendation for Android Development

Even when scoped for Android development, various code recommendation tasks can be performed. Wu et al. [17, 18] conduct a qualitative review on tens of code recommendation techniques that may be applicable, and identify various limitations that may need improvements for Android development. Among various issues for improvement, they briefly mention that the component-based and event-driven programming nature of Android development needs tailored code recommendation techniques to integrate callbacks into recommendation results properly. We concur with the observation and think this is important for useful API recommendations for Android and other event-driven programming frameworks. Thus, we in this paper explore ways to recommend not only functional APIs but also event callbacks.

3.2. Functional API Recommendation

Thung et al. [1] recommend functional APIs based on the similarity between textual API descriptions and the given textural descriptions of a feature request. Yu et al. [2] also recommend functional APIs by utilizing the similarity between more semantic information about APIs and the given textual queries. Thung et al. [19] recommend Web APIs by taking in a project profile and ranking the APIs "personalized" for the project. Rahman et al. [3] utilize the crowdsourced knowledge of StackOverflow to identify keyword-API associations and construct more informative API descriptions for search.

The essential idea of their work and our paper is all based on a similarity measurement of certain "descriptions", while we utilize different data sources for creating API descriptions and recommend different kinds of APIs. There are also studies that do not rely on API descriptions and use techniques different from ours for API recommendation. For example, Xie et al. [20] use multi-relation models to recommend web APIs. Ramírez [21] develops an API recommendation system using collaborative filtering along with frequent itemset mining techniques. Shi et al. [22] utilize project topics and feature-interface graphs to recommend APIs. McMillan et al. [23, 24] use graph-based matching to identify relevant functions (not limited to library APIs) and their usages. Chan et al. [25] optimize the search algorithm for huge API graphs to improve recommendation results.

3.3. Library Recommendation

Closely related to but different from API recommendation is library recommendation, which recommends a library (containing many APIs) for development. For example, Thung et al. [26] recommend libraries by using association rule mining and collaborative filtering techniques. Chen and Xing [27] recommend similar libraries for development in different languages. McMillan et al. [28, 29] recommend similar applications based on a given application. How they measure "similar" or "different" libraries is then different from how we match APIs.

3.4. API Documentation Recommendation

Another closely related area of work is to recommend or infer documentation or properties about the identified APIs and libraries. E.g., Robillard and Chhetri [30] have developed a tool for information filtering and discovery that can recommend relevant document fragments to help developers understand selected APIs better. Guerrouj et al. [31] generate summaries of library

identifiers based on relevant posts on StackOverflow. Nazar et al. [32] further provide a survey of various summarization techniques for various software artifacts and their applications, including documentation generation.

There are also many studies on inferring the properties of APIs. Robillard et al. [33] provide a comprehensive survey of over 60 techniques that can infer different kinds of properties. Our work does not infer API documentation or properties, but recommends which APIs to use. On the other hand, the two lines of work may complement each other: API recommendation may utilize the additional documents and properties generated for each and every API to find better matching results; documentation and property recommendation or inference may be more targeted convenient for developers to use if there are already APIs automatically recommended for them. This combination may be an aspect that moves research on on-demand developer documentation [34] forward.

3.5. API Usage Code Generation

It would be even better, after recommending certain APIs to developers, to recommend or generate and visualize sample code usages of the APIs so that it may be easier for the developers to understand and reuse the sample code in their code [4, 5, 8–12, 35–51], and even auto-complete developers' code or fix bugs using the APIs and proper parameters based on developers' coding contexts [6, 7, 52–58]. There are studies that aim to achieve these goals, but they have not been adapted for generating code samples for event-driven programs, where control flows and call relations are implicit and functionalities need to be implemented by not just invoking functional APIs but also overriding suitable callbacks. Although we believe it will be possible to adapt those techniques for our recommendation results to generate sample code too, we in this paper have not explored that and leave it for future work.

3.6. Various Data Sources for Recommendation

A variety of data sources may be utilized for code recommendation. Our study in this paper collects data from Android official tutorials and SDK documents, which may be arguably the most reliable set of reference materials for APIs and may help developers understand the design, rationale, usages, and potential pitfalls of the APIs better. Related work has utilized many other kinds of data for constructing API description databases (e.g., API usages in internet-scale source code base, questions and answers from Stack-Overflow, input/output specifications, execution traces and profiles, etc.).

While utilizing more variety of data on the internet may help to improve the comprehensiveness of the search results, it may also bring many challenges associated with heterogenous and possibly conflicting data and an overwhelming number of matching results. Our study in this paper does not address those challenges, while focusing on demonstrating the importance of recommending callbacks and using the official Android documents without the need of large dataset or heavy processing. Our future work will incorporate more data sources into our recommendation engine and integrate code usage generation for more comprehensive and usable API recommendation results, towards a more holistic recommendation system [59].

4. Approach

4.1. Overview

The overall framework of our approach is shown in Figure 3, following [13]. We know that developers often bring up questions about how certain functionality can be implemented by using existing APIs. Data sources like tutorials can give instructions and examples to show how certain APIs are meant to function, which can be used to answer such questions. Our work aims at bridging the gap between the questions and the data sources. We extract APIs and the descriptive texts that are closely related to the APIs, and recommend APIs whose related descriptions are most similar in semantics to the input question.

The work flow of our approach consists of two phases. The first phase refers to the construction of our databases for recommendation, and the second phase refers to the recommendation of functional APIs and event callbacks and their combinations.

In the first phase, we build our databases by processing and extracting information from the data of Android tutorials and SDK reference documents. Firstly, we collect raw data from *Android Tutorials* and *Guides*[4] (simply called *Android Tutorial*) and *Android SDK Reference*[5] from the official *Android Developers* website[6], which are in the form of HTML pages. Then we process the HTML files, discard irrelevant information, and store relevant information into XML files in our pre-defined structure for further processing. With XML

[4]`https://developer.android.com/guide/`
[5]`https://developer.android.com/reference/packages`
[6]`https://developer.android.com/`

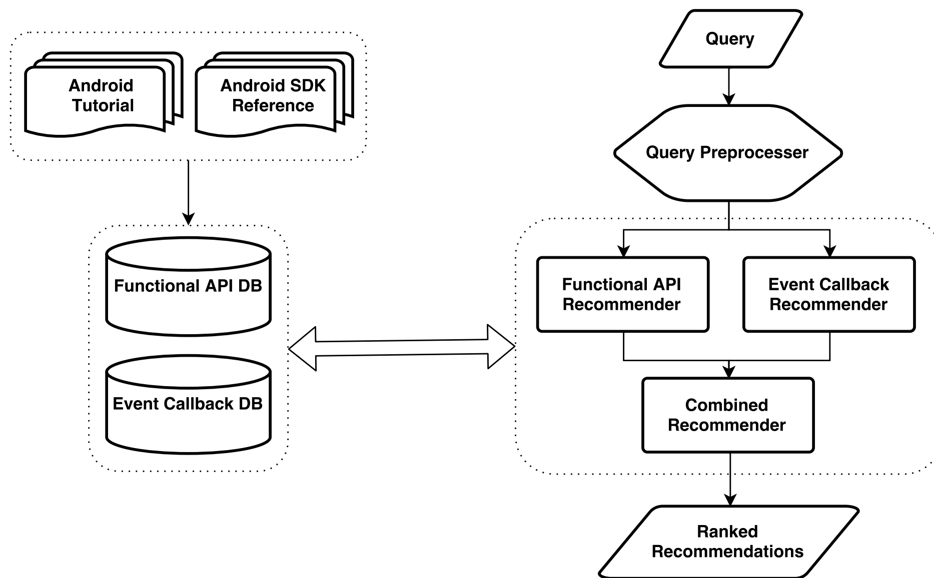


Figure 3: The Framework of Our Approach

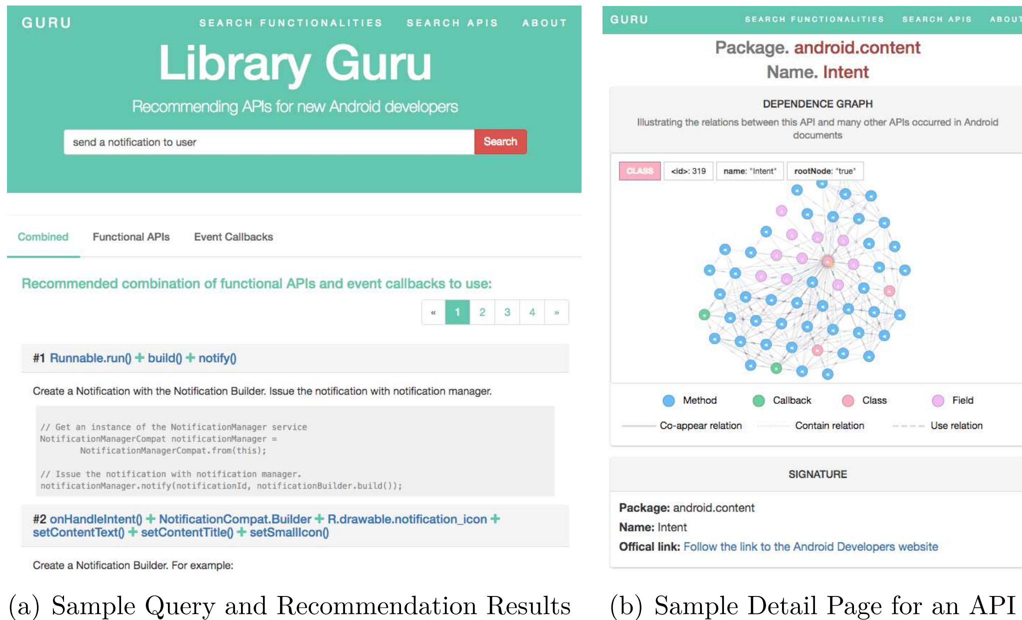
files of a uniform format, we extract correlations among APIs and descriptive texts, and store such correlations into our databases for recommendation.

In the second phase, we recommend functional APIs and callbacks from our databases according to a given query. Our tool takes as input a textual query which can be a sentence or paragraph to describe the needs to implement certain functionality, and returns a list of relevant functional APIs and callbacks accordingly. Before receiving any queries upon boot-up, the tool first initializes an in-memory search space using the databases set up in the first phase. For each query, the tool looks up the databases for descriptive texts that are most similar to the query, and outputs corresponding recommended APIs to which the descriptive texts are closely related.

4.1.1. Tool Usages

We have implemented a prototype of the approach in a tool, named LibraryGuru [13]. In addition to the backend command line interface for the tool, we have developed a web interface for LibraryGuru where users can easily input their queries in English into the search box on the web interface to retrieve recommendations.

Figure 4 illustrates the graphical user interfaces of LibraryGuru. Figure 4(a) indicates the recommended results for a query "send a notification to



(a) Sample Query and Recommendation Results (b) Sample Detail Page for an API

Figure 4: LibraryGuru Web User Interfaces

user". The top-1 result in the "Combined" tab indeed shows how a developer could implement the functionality with sample code: it can be done by overriding the `run` method in a `Runnable` thread that builds a notification and notifies a receiver by invoking the `build` and `notify` APIs. A user can also click the links on the API names for more details. Among the API signature and sample usages, the API detail page as shown in Figure 4(b) shows the correlations contained in our backend databases for the `Intent` class, in the form of a dependence graph.

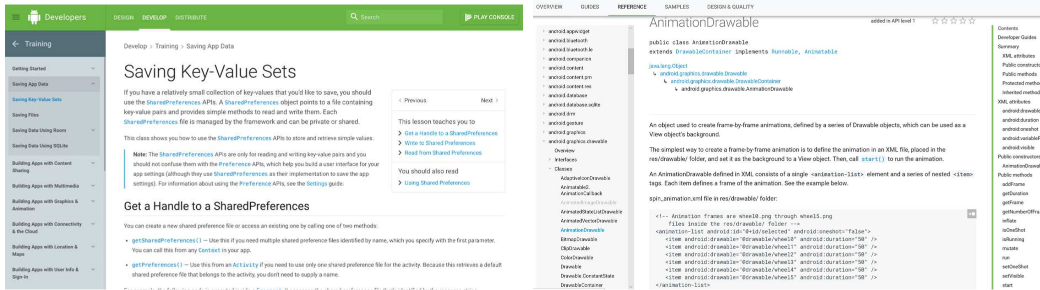
4.2. Data Collection and Transformation

In order to recommend related APIs that conform to the queries, we first build databases that store co-relationship pairs of APIs and descriptive texts that are related to the APIs.

4.2.1. Data Sources

We choose two data sources, *Android Tutorial* and *Android SDK Reference* from the official *Android Developers* web site, to build our databases.

- *Android Tutorial* is the "*Training*" or "*Guides*" subsection of the "*Develop*" or "*Docs*" section on *Android Developers* web site to introduce



(a) Sample Android Tutorial Page (b) Sample SDK Reference Page

Figure 5: Sample Tutorial and Reference Web Pages

developers different Android APIs (e.g., packages, classes, methods of classes). This Tutorial covers different scenarios and requirements about almost every aspect of Android development; the Tutorial is roughly divided into several main sections, and then further divided into detailed parts, each page only covers some implementations of a very specific functionality (e.g., Saving Key-Value Data Sets as shown in Figure 5(a)[7]). Most pages, according to the topic of the page, tell readers what APIs to use to achieve a certain goal, and there are often some snippets for demonstrating the usage of APIs. This feature enables us to find plenty of co-relationship pairs.

- *Android SDK Reference* is the reference document for all Android APIs. In Android SDK Reference, all *Packages*, *Classes*, *Interfaces* and other items (e.g., *Enums* and attributes) are indexed by package names. On each page of a Class or an Interface, it lists all APIs that can be used by the Class or Interface. Moreover, most of the Packages, Classes/Interfaces and APIs like methods have short or long descriptive texts explaining the usage of these API entities, which can be utilized to create co-relationship pairs. Figure 5(b) shows a sample reference page for the `AnimationDrawable` class[8], which explains basic usages of the class and provides a list of attributes and methods available from the class.

[7] `https://developer.android.com/training/data-storage/shared-preferences`

[8] `https://developer.android.com/reference/android/graphics/drawable/AnimationDrawable`

We observed that the sentences used in Android SDK Reference are rather lower-level, containing more technical terms. The descriptive texts in Android Tutorial, considering its purpose to teach developers to use Android APIs to implement different functionalities, are more of the same semantics level as the potential queries that may be asked by developers and supposed to be handled by our recommendation engine. Therefore, we use the data collected from Android Tutorial serve as our first-choice data source.

4.2.2. Data Downloading and Filtering

We download all the pages from our two data sources, which are in the form of HTML files. Among all the downloaded files, we discard the files that have no use in our case and keep the useful ones. For Android Tutorial pages, we only take pages that focusing on specific use cases as useful pages, therefore, pages such as index pages of each main section are filtered. Moreover, as we aim at APIs, pages that are more concerned about program designs than API usages are discarded. For Android SDK Reference pages, we remove indexing pages and other pages that are not in our interest.

4.2.3. Data Transforming and Splitting

With filtered HTML files, we further transform them into XML files which are easier to manipulate, and split them into even smaller XML files from which we can extract desired information to build the databases.

As we observe and analyze the filtered HTML pages, we find that the contents of a single page have a hierarchical structure according to various semantic levels of the contents, which can also be aligned to various HTML tags associated with the contents (e.g., contents in <h1> tags often refer to the headline of the whole page and those in <h2> tags often are headlines of the sub-parts of the page). Therefore, we parse the HTML file to get all the contents, and rebuild the *Document Object Model* (DOM) trees according to the hierarchy the HTML file is supposed to have. We save the new DOM tree in an XML file which we refer to as a *Raw XML* file. A *Raw XML* file contains as much raw information as its corresponding HTML file.

Figure 6 is a shortened sample snapshot of the *Raw XML* files. This example is to show how to create screen sliding effects using the `ViewPager` APIs as indicated by the <h1> tag, under the overall topic of “Adding Animations” as indicated by the <h0> tag. This example also has various sub-parts as indicated by the multiple <h2> and <h3> tags, such as “<h2>Customize the Animation with PageTransformer” and “<h3>Zoom-out page transformer”.

```
- <raw_xml>
  - <h0 description="How to add transitional animations to your user interface.">
      Adding Animations
    - <h1>
        Using ViewPager for Screen Slides
      - <APIs>
          <API href="https://developer.android.com/reference/android/support/v4/view/ViewPager.html">ViewPager</API>
          <API href="https://developer.android.com/reference/android/support/v4/app/Fragment.html">Fragment</API>
          <API href="https://developer.android.com/reference/android/app/Fragment.html#onCreateView(android.view.LayoutInflater,
          android.view.ViewGroup, android.os.Bundle)">onCreateView()</API>
          ...
        </APIs>
      - <p>
          Screen slides are transitions between one entire screen to another and are common with UIs like setup wizards or slideshows.
          This lesson shows you how to do screen slides with a
        - <code>
            <a href="https://developer.android.com/reference/android/support/v4/view/ViewPager.html">ViewPager</a>
          </code>
          provided by the
          <a href="https://developer.android.com/tools/support-library/index.html">support library</a>
          ... Here's what a screen slide looks like that transitions from one screen of content to the next:
        </p>
        <p>...</p>
      - <pre>
          <!-- fragment_screen_slide_page.xml --> <ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
          android:id="@+id/content" android:layout_width="match_parent" android:layout_height="match_parent" > <TextView
          style="?android:textAppearanceMedium" android:padding="16dp" android:lineSpacingMultiplier="1.2"
          android:layout_width="match_parent" android:layout_height="wrap_content" android:text="@string/lorem_ipsum" />
          </ScrollView>
        </pre>
        <h2>...</h2>
      - <h2>
          Customize the Animation with PageTransformer
        - <p>
            To display a different animation from the default screen slide animation, implement the
          - <code>
              <a href="https://developer.android.com/reference/android/support/v4/view
              /ViewPager.PageTransformer.html">ViewPager.PageTransformer</a>
            </code>
            interface and supply it to the view pager. The interface exposes a single method, ...
          </p>
        - <h3>
            Zoom-out page transformer
          - <p>
              This page transformer shrinks and fades pages when scrolling between adjacent pages. As a page gets closer to the center
              it grows back to its normal size and fades in.
            </p>
          - <pre>
              public class ZoomOutPageTransformer implements ViewPager.PageTransformer { private static final float MIN_SCALE
              = 0.85f; private static final float MIN_ALPHA = 0.5f; public void transformPage(View view, float position) { int
              pageWidth = view.getWidth(); int pageHeight = view.getHeight(); if (position < -1) { // [-Infinity,-1) // This page is way
              off-screen to the left. view.setAlpha(0); } else if (position <= 1) { // [-1,1] // Modify the default slide transition to shrink
              the page as well float scaleFactor = Math.max(MIN_SCALE, 1 - Math.abs(position)); float vertMargin = pageHeight * (
              - scaleFactor) / 2; float horzMargin = pageWidth * (1 - scaleFactor) / 2; if (position < 0) {
              view.setTranslationX(horzMargin - vertMargin / 2); } else { view.setTranslationX(-horzMargin + vertMargin / 2); } //
              Scale the page down (between MIN_SCALE and 1) view.setScaleX(scaleFactor); view.setScaleY(scaleFactor); // Fade
              the page relative to its size. view.setAlpha(MIN_ALPHA + (scaleFactor - MIN_SCALE) / (1 - MIN_SCALE) * (1 -
              MIN_ALPHA)); } else { // (1,+Infinity] // This page is way off-screen to the right. view.setAlpha(0); } } }
            </pre>
```


Figure 6: Sample Raw XML File

For Android Tutorial, each *Raw XML* file usually contains an <h1> tag which is the root of the whole file, and under <h1> tags the file is divided into several sub-parts which are led by <h2> tags, and those sub-parts may also be divided into parts led by <h3> tags. The contents of <h1> tag often summarize the main Android techniques to be explained on this page, while the <h2> or <h3> tags among its children give a general idea on what each sub-part focuses on. Each part led by <h2> or <h3> tags may contains usages of APIs (mostly demonstrated by code snippets) and descriptive texts with them. We capture various information from the page about the APIs, such as the links to their detailed SDK documents in the <APIs> and <code> tags, store the descriptive texts for them inside <p> tags, and collect sample code uses or configurations of the APIs in the <pre> tags, as illustrated by various tags in Figure 6.

For Android SDK Reference, the *Raw XML* is a large XML file which contains all the data in the SDK documents. In this file, Packages, Classes or Interfaces and all the APIs such as public methods are stored in different tags which are organized by the hierarchy they are supposed to have (e.g., the direct parent tag of a Class is the tag of the Package of this Class).

To make the data more fine-grained, we want each of the data files to only contain information about a single certain functionality and very likely to contain the APIs that are related to this functionality. Therefore, we further split *Raw XML* files into smaller files, which we call *Raw Doc* files. For Android Tutorial, each *Raw XML* file is split into several *Raw Doc* files with each *Raw Doc* file only have one sub-part led by <h2> tag. For Android SDK Reference, each *Raw Doc* file contains a certain API with its brief description as well as the descriptions of its Class/Interface and Package.

In our implementation, we use *Beautiful Soup* [60] to parse all HTML and XML files, and *lxml* [61] to create the XML files for our needs. *Beautiful Soup* is a HTML and XML file parsing library with simple interfaces for retrieving and extracting information from DOM trees. *lxml* is also a useful library for processing HTML and XML files with stable ability to create XML files.

4.3. Extract Co-relationship Pairs between Texts and APIs

With fine-grained, well-structured raw data files, we are able to extract information from them and create our databases. We define the entry for our databases to be a pair of ***<Description, API>***. We extract such pairs from both Android Tutorial and Android SDK Reference. We define ***<Description, API>*** as an abstract data structure representing a pair of

a piece of ***Description*** and an ***API***, which we also call a ***Co-relationship*** pair. ***API*** refers to a list of functional APIs or callbacks, and ***Description*** refers to the descriptive texts that are related to the APIs in ***API***.

4.3.1. Extracting Co-relationship from Android Tutorial

For every page of Android Tutorial, the main topic of the page is written in the <h1> tag, and the page is divided by several <h2> tags indicating more specific sub-topics, and sometimes there are also <h3> tags to further split the page, therefore we can see a clear hierarchy from the page as mentioned in Section 4.2. Under every <h2> (or <h3>) tag, the sub-part will show some information about what functionality can be achieved by using what APIs. Besides, we also add an <h0> tag when building *Raw XML* files, which is the title of the section that contains the HTML page. Therefore, for the APIs that appear in this part, the corresponding *Description* would contain the headline information concatenated together, as shown below:

$$Description = h0 + h1 + h2 + h3\ (if any) \quad (1)$$

where "+" refers to concatenation operation of a string.

Further, to make co-relationship between *API* and *Description* more informative, we also manage to find information which is more specific to the target APIs, which is denoted by *Info*. Therefore, Equation 1 is refined to:

$$Description = h0 + h1 + h2 + h3\ (if any) + Info \quad (2)$$

There are two ways to find APIs and corresponding *Info* from Android Tutorial, and then extract *Co-relationship* pairs. In the pages of Android Tutorial, there are many texts that directly mention some APIs along with texts describing the information about the APIs. In addition, there are also code snippets following some descriptive texts to demonstrate the example of using certain APIs, and some of the code snippets even have comments on some key codes inside the snippets which explain the purpose of the line of code following the comments. Therefore, we can extract *Co-relationship* pairs not only from texts in the Android Tutorial, but also from the code snippets.

4.3.2. Extracting Co-relationship Pairs from Texts

In the original HTML files of Android Tutorial, for all the HTML tags that refer to texts (e.g., <p> tags which contain a paragraph), when an Android API is mentioned, it always appears with an <a> tag in which the

text is the name of the API and the *"href"* property points to the link of the API in Android SDK Reference, such as those <a> tags in Figure 6. In this case, with the APIs mentioned in the texts as *API* and the texts as *Info* in the corresponding *Description*, we can extract the *Co-relationship* pairs from the texts in Android Tutorial.

4.3.3. Extracting Co-relationship Pairs from Code Snippets

On Android Tutorial pages, besides texts describing APIs, there are often code snippets to illustrate the usages of the APIs, which are stored in <pre> HTML tags. Some of the <pre> tags contain configurations in the format of XML to set up properties related to the APIs (e.g., the first <pre> tag in Figure 6 defines a scrollable text view to display some texts); some other <pre> tags contain actual Java code that uses the APIs (e.g., the second <pre> tag contains sample code that shrinks and fades pages when users scroll between adjacent pages by implementing the `ViewPager.PageTransformer` interface). Depending on whether a code snippet contains comments or not, we have two ways to extract <*Description*, *API*> pairs.

We utilize *Eclipse JDT* [62] to implement a customized Java code parser. For each snippet in a <pre> tag that contains Java code, we use our parser to parse the potentially incomplete Java code. With the help of the parser, we can get the *Abstract Syntax Tree* (*AST*) of the snippet, from which we are able to get all kinds of AST nodes representing different syntaxes.

To extract *Co-relationship* pairs for *functional API*s, we first need to clarify what should be considered as a functional API. With our knowledge on Android development, the most common case to use APIs is to invoke some public methods in some instances of some classes or to invoke some static methods in some other classes. And to invoke methods, developers often need to create a new instance of certain class with various parameters, some of which can be pre-defined fields of the class. Therefore, we focus on looking for the three kinds of APIs mentioned above, which are public methods, classes, and fields (often constants). With our parser, we can find three types of AST nodes which are respectively correspondent to the three kinds of APIs, namely *MethodInvocation*, *ClassInstanceCreation* and *QualifiedName*. A *MethodInvocation* node refers to an invocation of a method; if such node is in the AST of a snippet, it means that some method is invoked in that snippet, such as the 3 calls to `setAlpha` in Figure 6. A *ClassInstanceCreation* node appears only when a new instance of a class is created, usually by the ***`new`*** keyword. By looking for these two kinds of

nodes, we can get what methods and classes are called or used in the snippet. *QualifiedName* refers to a *SimpleName* (an identifier) with a qualifier in the form of "*Name.SimpleName*", which matches our pattern to look for public fields (often constants). Intuitively, we only care about public fields that are used when invoking a method or instantiating a class, therefore, only *QualifiedName*s whose parent node is one of the above two nodes are extracted. The fields `MIN_ALPHA` and `MIN_SCALE` in Figure 6 are not public, although they are constants and used for invoking `setAlpha`, and we heuristically do not consider them relevant to `setAlpha`.

With our parser, we can also find out if there are any comments in the code snippet, and get all of them if they exist. If a line or several continuous lines of comments are found in the code, we check whether the comments are directly followed by any code. If there is any AST node of the three kinds above, we then collect the APIs represented by those nodes, and use the comments as the *Info* in the corresponding *Description*. For example, in Figure 6, the second <pre> tag contains multiple comments indicated by "//"; and then each comment is heuristically used as the description for the lines of code following the comment. E.g., the comment "// Scale the page down (between `MIN_SCALE` and 1)" is the description for the following two API calls `setScaleX` and `setScaleY`.

When no comments are found in the snippet, we find all the APIs of the three kinds to be the *API* of the *Co-relationship* pair. As for the *Info* in the *Description*, we search backward in the DOM tree of the Tutorial page to find the nearest tag containing textual descriptions (e.g., a <p>, <h3>, or <h2> tag) as we observe that snippets with no comment inside often follow some texts describing what the snippets are used for.

To extract *Co-relationship* pairs for *event callback*s, as event callbacks are methods which are supposed to be overridden by developers to implement their own functionalities within, we lay our emphasis on finding methods that are overridden in the code snippets. Based on the specification of *Eclipse JDT*, a *MarkerAnnotation* node in the AST refers to an annotation in the code. Therefore, to find event callbacks in the snippet, we look for *MarkerAnnotation* nodes which represent *`@Override`* annotations. For the event callbacks appearing in a code snippet, we use the contents of the nearest preceding textual tags as *Info* in the *Description*. With the help of our parser, we can also get the parent node of the overridden method which refers to the class in which the method is overridden. We can also get what is the extended

`Class` of the class or what `Interfaces` the class implements, and thus we are able to identify where the overridden method really comes from. For example, the second <pre> tag in Figure 6 contains code that "`implements ViewPager.PageTransformer`", which is a Java interface, and thus the public method `transformPage` implemented in the code is likely a callback that may be invoked by the framework at certain situations. Indeed, according to the tutorial page, application developers do not invoke `transformPage` directly; it is invoked by the underlying framework for each visible page whenever the screen's transition/scrolling happens. Thus, we identify `transformPage` as a callback API, and the nearest <p> tag texts and the <h3> tag texts ("Zoom-out page transformer" in Figure 6) are used as its descriptions.

Meanwhile, we can also get all the functional APIs used in the callback, which in turn helps us to infer some correlations between the functional APIs and the callbacks. For example, the callback API `transformPage` in Figure 6 contains calls to various functional APIs, such as `getWidth`, `setTranslationX`, `setAlpha`, `Math.max`, `setScaleX`, etc., which indicates that these APIs may often be used together. By counting the co-appearances of one functional API and one callback, we are able to learn how often a functional API is used in a certain callback. The co-appearance information helps us to recommend both functional APIs and callbacks in combination.

Due to the semantic limitations of often incomplete code snippets and our parser, what is returned from the parser are only names of methods, classes or fields without sufficient type information. Our parser built on top of Eclipse JDT so far does not guarantee correct parsing of all the code snippets. When a code snippet cannot be parsed, our parser simply discards it. When a code snippet can be parsed, our approach tries to get more information about the APIs used in the snippet: for all APIs extracted from the Tutorial page, we collect all <a> tags in the page which links to an API in the Android SDK Reference to build a static lookup table. If any extracted API name matches the text of any <a> tag in the table, then it is very likely that the extracted API is the API that links to the <a> tag; therefore, we can get more information about the API, such as its package name or parameter types. For example, the first <code> tag in Figure 6 provides a <a> link to the more detailed page for the `ViewPager` class, and this information is also collected into the <APIs> tag in the raw XML file in Figure 6.

For API names that do not have a matching <a> tag, we treat them differently to build two databases accordingly. One way to deal with these API names is to remove them from the *API* list of the *Co-relationship* pairs,

in which we can get a database where all APIs in it have more detailed information and are all APIs in the Android SDK Reference. However, in this way the number of pairs extracted is rather too small (it was only 409 in our initial exploration) for API recommendation, and the average numbers of APIs of each pair is also not considerable. Another way is to keep these API names even though the only information about them is the name of the API, which enables us to extract as many pairs as possible. For example, in Figure 6, although the class `View`, the methods `setAlpha`, `setTranslationX`, `setScaleX`, etc. are used in the code, no explanations or links to other documents are provided, and we still keep these API names in the database, and only add descriptions for them if we find relevant descriptions when processing other tutorial or reference pages. In this way, the scale of the databases is more than twice as that with the first way: the numbers of *Co-relationship* pairs is also up to 1119. Therefore, we adopt the second way to build each *Co-relationship* pair. The advantage of this way is even more significant when we extract *Co-relationship* pairs for *event callback*s as most callbacks do not have a matching <a> tag in the lookup tables.

4.3.4. Extracting Co-relationship Pairs from Android SDK Reference

For Android SDK Reference, it is simple to get the API and its corresponding description of each *Raw Doc* file, and the descriptions of its Class/Interface and its Package. Therefore, the *API* in the *Co-relationship* would be the API in each *Raw Doc* file, and its corresponding *Description* is composed of the descriptions of the API and its Class or Interface and its Package. When no description can be found for any of the three, we simply use their names as descriptions. The data scale of Android SDK Reference is quite large, therefore we only extract *Co-relationship* pairs for public methods and fields of Classes and Interfaces, which still gives us a total of 52781 pairs.

According to our knowledge, most of event callbacks regarding Android development begin with the word *"on"*, for example, the "`onStart()`" method is invoked when an `Activity` starts. Therefore, to get *Co-relationship* pairs for *event callback*s, we simply select all pairs in which the API is a method and its camel case name begins with *"on"*. In this way, we can get 2469 *Co-relationship* pairs for *event callback*s. The rest 52312 pairs are considered pairs for *functional API*s.

4.4. Compare Similarity between Query and Description

After we extract all *Co-relationship* pairs, we serialize them into files to serve as our databases. When our recommendation tool starts, it will first load the files and take out all the *Co-relationship* pairs. Each input query fed into the tool is first preprocessed with Query Preprocessing. For every query and *Description* of the *Co-relationship* pairs, the tool applies some *General Preprocessing* steps of NLP to the texts, and computes the similarity between them. According to the similarity between the query and all *Description*s in the databases, the tool recommends APIs in the *API* of the corresponding *Co-relationship* pairs whose *Description*s are most similar to the query.

4.4.1. General Preprocessing

To compare the similarity between a query and the *Description*s stored in the databases, we preprocess the query and each *Description* with common NLP preprocessing steps.

We use *bag-of-words* [63] model, a simple and popular model to represent a text as a bag (multiset) of its words, regardless of word order. For each document, we first remove punctuation and tokenize it, which will break the text into word tokens. After tokenization, we remove some stop words that are noisy to the document such as pronouns like "the" and link-verbs like "are". Last but not least, we use Porter Stemmer to find stems for each word and replace the original word with its stem word.

Due to the characteristics of our data, there are always some domain-specific words in the texts, most of which are inline Java names in the form of *"Camel Case"*. For this kind of words, we find them before stemming, and split them into several words according to the capital letters and add the words in the document as well as keeping the original word.

All the preprocessing are implemented on top of *NLTK* [64], a toolkit for natural language processing in python.

4.4.2. Term Frequency-Inverse Document Frequency

With the help of all the preprocessing, a text document is represented as a bag of stem words. To take one step further, we use *vector space model* [65] and *Term Frequency-Inverse Document Frequency* (*TF-IDF*) weighting to represent each document. According to vector space model, a text document is represented as a multi-dimension vector of which, in our case, every dimension represent a unique word in the corpus (i.e., all documents, in our case, all

Descriptions in the database) and the value of the dimension is the *TF-IDF* of the word in this document. Term Frequency (*TF*) [66] indicates how often a term (in our case, a word) appears in a document, which is calculated by dividing the number of the word by number of all words in the document. However, term frequency alone can not tell how strong the relation is between a word and its document. For example, the word "word" could appear many times in each document, which makes the TF for the word quite high for every document, but the word itself hardly contains any useful information. Therefore, we use Inverse Document Frequency (*IDF*) [67] to balance. *IDF* indicates how common or rare a term is across the whole corpus. For a term t in the corpus C, the Document Frequency (*DF*) is the number of documents that contain the term. The *IDF* can be calculated in different ways, one of which is: $idf(t, C) = \log \frac{|C|}{df(t, C)} + 1$, where $|C|$ is the total number of documents in corpus C and $df_C(t)$ is the document frequency of term t. In our approach, we use *Smooth IDF*, which adds 1 to document frequency, as if an extra document was seen containing every term in the collection exactly once. *Smooth IDF* can avoid zero division, and is calculated as follows: $Smooth_idf(t, C) = \log \frac{1 + |C|}{1 + df(t, C)} + 1$. With *TF* and *IDF*, the *TF-IDF* weight of a term is computed in the following way: $tfidf(t, d, C) = tf(t, d) \cdot Smooth_idf(t, C)$.

4.4.3. Cosine Similarity

To compute the similarity between two documents with their corresponding *TF-IDF* vectors, we use *Cosine Similarity* to indicate how similar these documents are. Cosine similarity measures the orientational differences between two non-zero high-dimensional vectors, ignoring the differences in their lengths, which is suitable for our setting where the queries and the API descriptions in our databases can have very different positive sizes and the value for each word dimension is non-negative. Also, some studies [68, 69] have shown that, for the tasks of information retrieval and document clustering, cosine similarity often performs better than many other metrics, such as Dice coefficient, Jaccard coefficient, Euclidean distance, etc. Although there exist potentially better performing similarity metrics, such as TS-SS (the Triangle's area Similarity and Sector's area Similarity [70]), we do not focus onto the task of selection of "best" metrics in this paper and only leave that for future work and tool refinements.

Given two vectors of A and B of size n, the cosine similarity is represented as: $Cosine_Similarity = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$, where A_i and B_i refer to the ith element of A and B respectively.

4.4.4. Special Query Preprocessing

Before a query is preprocessed by the *General Preprocessing* steps, we first conduct some special processing on the query to de-noise the query and obtain as much as semantic information from it.

Although we are using stemming to unify the words into a unique root form in order to match as many as related words between documents, we are not able to match a word with synonyms. Therefore, we use *WordNet* [71], a lexical database for the English language to find synonyms for words selectively according to their *part of speech*. We replace the original words with the synonyms to create new sub-queries and input them into our tool, get all the results and re-rank them.

4.5. Recommend Functional APIs and Event Callbacks

We recommend APIs based on the similarity score of the query and the *Description* documents in our databases. For recommending functional APIs or event callbacks alone, the *Functional API Recommender* or *Event Callback Recommender* simply finds the most similar *Description* documents in the corpus, and returns the APIs in the corresponding *Co-relationship* pairs.

We also try to recommend both functional APIs and callbacks in which the recommended functional APIs are supposed to be used, which we call *combined recommendation.* For combined recommendation, we utilize the co-appearance of functional APIs and callbacks mentioned above, which we refer as *co-appearance databases.* For a combined recommendation, the query is first fed into *Functional API Recommender* to get recommended functional APIs. We use a `Counter` to record co-appearing event callbacks: with each functional API in the recommendation, we try to find all the co-appearing callbacks in the *co-appearance databases*, and put them in the `Counter`. In the end, if the `Counter` is not empty, it means that at least one functional API has co-appearing callbacks in the *co-appearance databases*, we select the callback with the highest count, which indicates that the callback co-appears the most frequent with the recommended functional APIs. For example, given a query, if the *Functional API Recommender* returns a recommendation with three APIs, which are A_1, A_2, A_3, and in the

co-appearance databases we find that the co-appearing callbacks of the three APIs are $\{C_1, C_2, C_3\}$, $\{C_1, C_3, C_4\}$, $\{C_3, C_5\}$ respectively. Then the `Counter` would be like $\{C_3 : 3, C_1 : 2, C_2 : 1, C_4 : 1, C_5 : 1\}$, and our recommending callback would be C_3. If no co-appearing callbacks are found in the *co-appearance databases* for any of the recommended functional APIs, we then simply use the *Event Callback Recommender* to recommend callbacks.

5. Empirical Evaluation

In this section, we evaluate the effectiveness of our recommendation engine for recommending relevant APIs for a given query in the form of textual description of a functionality. We are particularly interested in seeing how callback recommendations affect the overall recommendation results.

5.1. Research Questions

Specifically, we evaluate our techniques and tool for the following questions:

RQ1 What may be the optimal settings for our recommendation engine?

RQ2 What may be the precisions and recalls of our recommendation engine for both functional and callback APIs?

5.2. Settings

In our evaluation, we evaluate our approach on 1531 questions collected from StackOverflow regarding Android development that are mentioned in section 2.1. We randomly pick a portion, a total of 370 questions from the 1531 questions as a test set, to try different choices of technical settings in our approach and then use the setting with the best results on the test set for the larger evaluation on all of the 1531 questions.

For each question, we select informative texts from the question as the query input for our tool, and get the *top-N* recommendation(s). Then we compare our recommendations with the actual answers to the question on StackOverflow that meet the conditions that the *vote* must be at least 1 and there are code snippets in the answer with comparable functional or event callback APIs inside. If one recommendation is matched with any of the valid answers, then this recommendation is considered matched with the question.

5.2.1. Evaluation Metrics

For evaluation of the recommendations of functional APIs or callbacks separately, we compare the API list of each recommendation with each answer to the question on StackOverflow: if an API in the recommendation list is matched with an API in the valid answer, then the API recommendation will be considered as a matched recommendation. For evaluation of combined recommendations of functional APIs and callbacks, we compare pairs of <callback, functional API used in the callback> from the recommendation list with each answer, if any pair matches some functional API and callback in the answer, we then consider the recommendation as a matched recommendation.

We use three metrics to measure the performance of our tool, *Precision@N*, *Mean Average Precision*, and *Recall@N*.

Precision@N (P@N) is calculated by dividing the number of questions that have at least one matched recommendation among the *top-N* ranked recommendations by the number of all query questions. I.e., $P@N = \frac{|Q_N|}{|Q|}$, where Q is the set of all questions, $|Q|$ is the size of Q, and $Q_N = \{q|q \in Q \wedge \exists r$ in the top-N recommendations for q, *s.t.*, r is a relevant answer for $q\}$. To determine whether a recommendation r is relevant for q, we assume the answers with at least one up-vote for q on StackOverflow were correct, and then, if an API name recommended in r matches an API name used in a code snippet in the answers for q, we assume r is a relevant answer for q.

Mean Average Precision (MAP) is a popular metric for measuring recommendation algorithms in information retrieval. To calculate MAP, we first need to calculate AP (i.e., Average Precision) for each query. Average Precision takes the ranking position of relevant items in the recommended item list into consideration. For top-N items in a recommendation list, the equation of calculating Average Precision for a query q is as follows [72]: $AP = \frac{\sum_{k=1}^{N} (P(k) \times rel(k))}{number\ of\ relevant\ documents}$, where $P(k)$ is the *precision* at a cut-off rank k in the list and $rel(k)$ is a function equaling *1* if the document at the rank k is relevant to the query and 0 otherwise. The *number of relevant documents* refers to the number of all relevant documents in the corpus that are relevant answers for each query, which is, in our case, the number of all relevant APIs in the Android framework that can be used for implementing a functionality described in a query. However, this number is unknown in situations like ours where we do not have all possible answers for each query question. To handle such a problem when calculating AP, we optimistically

assume that all the top-N items in our recommendation lists that match an answer from StackOverflow for a query question form the total set of relevant APIs for the query. Then, we can estimate the AP for each query and the MAP can be calculated by averaging all APs: $MAP = \frac{\sum_{q=1}^{|Q|} AP(q)}{|Q|}$, where Q is the set of all queries.

To further compensate the inaccuracy in measuring MAP due to lack of ground truths in our situation, we approximate the measurements of ***Recall@N (R@N)*** for all questions, which estimates what is the ratio of all correct answers that are recommended by our tool. For this calculation, we also assume that each answer with at least one up-vote on StackOverflow were correct and that all the correct answers for each query question formed the total set of relevant answers for the query. Specifically, we calculate $R@N$ as $\frac{\sum_{q=1}^{|Q|} |R_N(q)|}{\sum_{q=1}^{|Q|} |A_c(q)|}$, where $R_N(q)$ is the set of relevant recommendations contained in our top-N recommendations for the question q, and $A_c(q)$ is the set of correct answers from StackOverflow for the question q. Note that, although many questions do not have any correct answer, even more questions have more than one correct answer on StackOverflow and each correct answer may provide more than one code snippet using different APIs for resolving the query question; thus, $\sum_{q=1}^{|Q|} |A_c(q)|$ is often much larger than $|Q|$.

Among the set of 1531 questions, every question has at least one answer that provides code snippets containing API usages, which are used to evaluate our *Functional API Recommender*. Meanwhile, there are 540 questions whose answers include snippets that have overridden callbacks, which we use to evaluate our *Event Callback Recommender* and *Combined Recommender*. In order to match a recommendation r to an answer from StackOverflow, we also apply the same code parser used for processing tutorial and reference data (cf. Section 4.3.3) to identify APIs used in the answer and compare the API names in r against the API names used in the code snippets in the answer. Due to the same limitations in the parser mentioned in Section 4.3.3, an answer was simply discarded if the parser cannot parse a code snippet in the answer post.

5.2.2. Databases

There are many databases to choose for both *Functional API Recommender* and *Event Callback Recommender*, which are different sets of *Co-relationship*

pairs extracted from Android Tutorial or Android SDK Reference in different fashions. For functional API recommendation, we have three sets from Android Tutorial, one is extracted from textual contents of Android Tutorial pages (with 1631 pairs), the other two are extracted from code snippets with different ways to handle APIs that do not have matching <a> tags in the API lookup table (409 and 1119 pairs respectively). We also have one set extracted from Android SDK Reference (52312 pairs). For callback recommendation, we have two sets, one from code snippets in Android Tutorial (286 pairs) and one from Android SDK Reference (2469 pairs).

According to our evaluation on the small test set of 370 questions, using the database with 1119 *Co-relationship* pairs from Android Tutorial snippets gives the best result for functional API recommendation and using the database with 286 *Co-relationship* pairs that are also extracted from Android Tutorial snippets gives the best result for callback recommendation. Therefore, we choose these two as the databases for our *Functional API Recommender* and *Event Callback Recommender* respectively.

5.2.3. Query Inputs

For each question, we have two choices to generate the input query for our tool. We can directly take the title of the question as our input, or we can include more information, adding the question body which contains further explanations of the question title. When adding question bodies, we also have two choices to reduce noises of the body texts by trimming information about codes in the bodies. One way is to remove all code snippets in the answer body, the other is to remove code snippets as well as inline codes in the body.

By evaluating with the test set of questions, we find that adding question bodies into queries yields better result, which can be explained that with enriched information the recommender can perform better in finding similar *Description*s. We also find that the result of removing both code snippets and inline codes from question bodies is better than that of only trimming code snippets, which is probably because that information about codes can introduce much noise for the actual focus and semantics of the queries.

5.2.4. Synonyms

As mentioned in section 4.4, we preprocess queries before tokenizing and stemming them, which is using synonyms to create similar queries.

By the evaluation with the test set, we find that the result of using synonyms is actually no better than without using synonyms. The possible

reason is that the synonym sets of *WordNet* were trained on general-purpose documents and do not apply well to Android development. This calls for future work to train domain-specific synonym sets for Android.

5.2.5. Stanford Parser for NLP

Sometimes in the queries users may not only ask about the functionality they require, but also implicitly ask about the callback methods where they want to implement the functionality by adding clauses like "when/after/before some event happens". Therefore, we try to parse the query to find such clauses. We utilize the *Stanford Parser* developed by Stanford NLP Group[9] to parse queries. The Stanford Parser is a PCFG (i.e., Probabilistic Context Free Grammar) parser [73], it can build a phrase structure tree from a sentence, in which we can easily find all phrases and clauses. If there is any clause that involves events, we get the clause and mark it as *callback query*, and the rest part are marked as *functionality query*. In this case, the two queries are handled by *Functional API Recommender* and *Event Callback Recommender* respectively.

When the question body is added into a query, this parsing is disabled due to time consumption and potential parsing errors of paring multi-sentence texts. Therefore, the parser is only enabled when the query contains just one sentence, which, in our evaluation, is often the case when we only use question titles as input, as question titles usually consist of one sentence. However, questions asking both functional APIs and event callbacks only take a small percentage in our question set (only 14 in 370 questions, and 48 in all 1531 questions), which makes whether using Stanford Parser or not hardly a factor to cause differences in the evaluation results.

[9] `https://nlp.stanford.edu/software/lex-parser.shtml`

Answer summary for **RQ1**: Based on a test set data, we observed that (1) Using tutorial documents are helpful in identifying more accurate correlations among various APIs and the various functionalities they may be used to implement, while API SDK documents alone could be too generic and low level to provide meaningful descriptions that can match functional query questions from users; (2) Using natural language parsers to enrich query questions with more detailed functional and event descriptions is useful in improving query matching; (3) Reducing noise in API databases and query questions, such as avoid using incorrect synonyms and removing questionable code in questions, is useful in improving query matching.

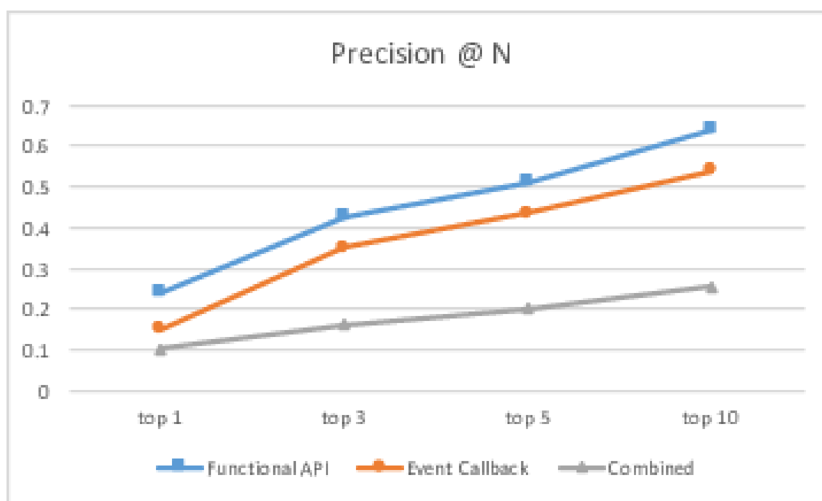
5.3. Results on Realistic Questions

According to the discussions about different choices of settings where we use 370 questions to test, we finally use the following settings for our evaluation on all of the 1531 questions. On choosing databases, we use the 1119 functional API *Co-relationship* pairs extracted from Android Tutorial snippets as our *Functional API Database*, and the 286 event callback *Co-relationship* pairs extracted from Android Tutorial snippets as our *Event Callback Database*. On choosing query inputs, we use both the title and the body but discard both code snippets and inline codes. On whether using synonyms, we do not use synonyms yielded from *WordNet* to generate similar queries, but only use the original query as the query input. On whether using Stanford Parser, as we choose to add the question body into the query, the parser is disabled.

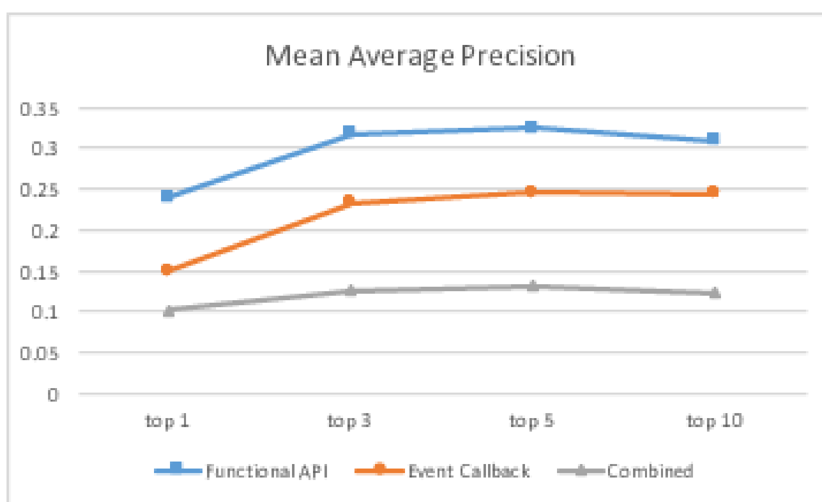
Using the above settings, we evaluate our tool on the 1531 questions that involve functional APIs, 540 of which also involve callbacks. We evaluate the top-N recommendations of our *Functional API Recommender* for the 1531 questions, where N is 1, 3, 5 and 10 respectively. We evaluate the top-N recommendations of our *Event Callback Recommender* and its combination with *Functional API Recommender* for the 540 questions, where N is also 1, 3, 5 and 10 respectively.

The *Precision@N* scores of our evaluations are shown in Figure 7(a), and the *Mean Average Precision (MAP)* scores are shown in Figure 7(b).

According to our evaluation results, the precisions of functional API recommendation can be up to 50.9% at top 5, and 63.9% at top 10, while the precision score of event callback recommendation is 43.5% at top 5, and 53.9% at top 10 respectively. The mean average precision scores of our tool for recommending functional APIs and event callbacks are 32.5% and 24.6%



(a) Precision@N



(b) Mean Average Precision

Figure 7: Precisions of Top-N (N=1,3,5,10) Recommendations

at top 5 respectively. Recommending functional APIs have higher precisions than recommending callbacks, possibly due to the fact that both functional APIs and query questions have richer descriptions about the functionality and it is more specific to match a question against functional APIs than to match callbacks. These results are arguably reasonable, considering that we

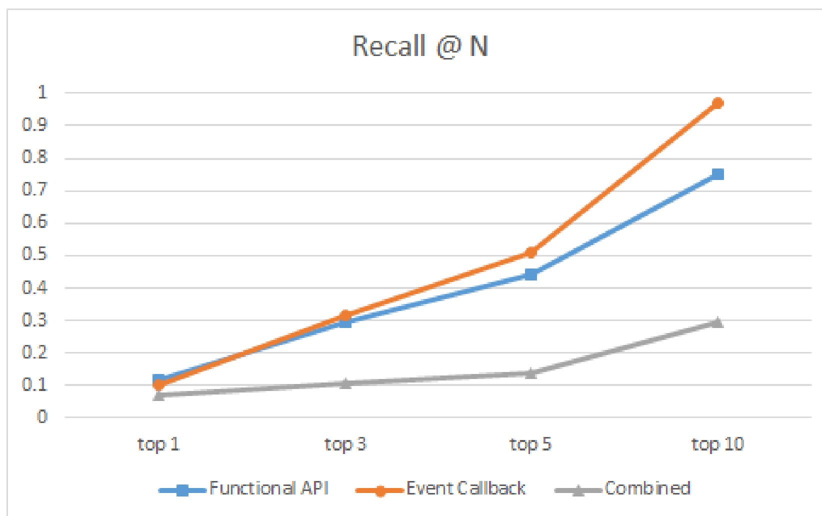


Figure 8: Recalls of Top-N (N=1,3,5,10) Recommendations

are only using relatively small sets of data sources for recommendation and that the way we collect questions does not ensure all questions really provide the queries that our recommendation system could handle.

As for combined recommendations, the precision score is 20.2% at top 5 and 25.6% at top 10, while the mean average precision is 13.1% at top 5 and 12.4% at top 10. A possible key reason for the much lower precisions for the combined recommendation is the vast number of possible ways how functional APIs can be used together with callbacks: We observed that many functional APIs can in fact be used flexibly with various callbacks, as callbacks can be override to contain almost any code, to implement a functionality in many different ways; it can be a guessing game when the question itself is unclear what events or under what situations it wants the functionality. It also indicates that our algorithm for combined recommendations has much space for improvement; future work can be considered to improve the matching by identifying more relevant correlations among functional and callback APIs from real code bases, and using an interactive mode to suggest developers to provide more contexts for their questions.

Figure 8 shows the *Recall@N* scores, based on the estimated calculations described in Section 5.2.1. The recalls range from 11.9% to 74.9% for *Functional API Recommender*, 10.1% to 97.1% for *Event Callback Recommender*, while ranging lower from 6.9% to 29.6% for their combinations. The recall

of top-1 functional API recommendations is slightly higher than that of callbacks. But the recalls of callbacks become higher when the number N of recommendations increases, likely due to the fact that the total number of callbacks is much fewer than functional APIs to choose from.

To give a glimpse of the time efficiency of our tool, for top-5 recommendation in our evaluation, the average time of generating recommendation results per query for functional API recommendation, event callback recommendation and combined recommendation is 41.4ms, 27.4 ms and 83.6ms respectively.

Answer summary for **RQ2**: Based on the evaluation on the 1531 questions, we observed that (1) our recommendation engine is efficient in both the phases of generating API databases and matching queries; (2) the precisions and recalls of both functional and callback API recommendations are promising; recommending functional APIs has higher precisions than recommending callbacks as their functional descriptions are richer and easier to match than callbacks, while recommending callbacks has higher recalls when the number of recommendations returned from the engine increases, probably because there are much fewer callbacks than functional APIs to choose from; (3) the combined recommendations of functional and callback APIs have much lower precisions and recalls, due to a vast search space where they can be combined in many different ways. Future work to utilize more code bases and contexts where developers give their questions may help improve recommendations.

5.4. Discussion & Threats to Validity

Many choices of design, implementation, and parameter settings in our API recommendation engine can affect its performance significantly. While our limited prototype and settings have shown that recommending event callbacks together with functional APIs can be promising, we still have much to do to reduce various threats to validity and improve the Android API recommendation engine for practical uses.

5.4.1. Data Sources

Intuitively, the more data for constructing the API description databases, the better the recommendation results may become. However, we observed the contrary results while incorporating Android SDK documents together with the development tutorials. Particularly, we tried to generate callback description databases from Android SDK documents for query matching, but the results were worse than using the callback description database generated

from tutorials. Even though SDK documents are of high quality in explaining low-level technical details, they may be only indirectly related to the functional requirements faced by developers, and do not fit well with the right semantic level of the requirements, causing mismatches in the recommendation results.

To avoid potentially conflicting and heterogenous descriptions for APIs, we in this paper do not utilize “big data” (e.g., existing answers in online forums such as StackOverflow), as the data we used are considered to be official Android documents of high accuracy, and may match many realistic usage scenarios better where many programming frameworks and libraries may have little “big data”.

Also, the tutorials and references can evolve, just like the APIs themselves and their usages evolve. It will be interesting future work to utilize differences among multiple versions of the tutorials and references to recommend APIs, especially when developers need to replace some APIs used previously with new ones. Dealing with changing documents with many versions can be more challenging but helpful for software evolution and maintenance.

All of these imply that the quality and abstraction level of the data sources play an important role for better recommendation results, likely more important than the quantity of data. As mentioned in the related work, utilizing more data more properly will be a long-term progressive goal that our study in this paper does not address, but will in the future.

5.4.2. Limited Code Parsing and Analysis

Our parser to process code snippets in tutorials, references, and StackOverflow posts is built on top of Eclipse JDT, which may not be able to parse incomplete code correctly or miss semantic information such as types. Although we have incorporated heuristics to identify classes, methods, fields (cf. Section 4.3.3), there are cases where our parser cannot parse a code snippet or extract API names, and are simply discarded. Future work that utilizes more adaptive and error-tolerant parsing [74, 75] and semantic parsing and fixing [76–79] can be useful to overcome such limitations.

There are also different kinds of information available in the Android tutorials and references that our parser cannot process, such as architecture or flow or state diagrams showing how different Android components are organized or how some APIs work together, and animations or videos demonstrating the effects of some APIs. Image and multimedia processing techniques may also be needed in future to utilize more information to improve the capabilities of API recommendation engines.

5.4.3. API Description Generation

To associate APIs with proper descriptions, we use simple structural and spatial association rules based on their occurrences in tutorial pages, as described in Section 4. The rules are still different from the semantics of APIs, and may potentially have many mis-associations, leading to inaccurate API recommendation results. It remains a challenging and active research topic in the areas of natural language processing and information retrieval to extract semantically related terms. It will be interesting to adapt many of those techniques to generate more accurate "<Description, API>" pairs.

Further, it is often needed to recommend multiple APIs together for developers to implement a functionality. Our description generation rules can generate descriptions for some such correlated APIs based on their co-occurrences in the HTML pages. However, for such cases, the generated descriptions may be too low level, explaining the functionality of individual APIs only, without higher-level, sufficiently summarized information for the set of APIs as a whole. This is also a semantic gap we observed frequently between the generated descriptions and the developers' queries in our study and many other code recommendation studies. This is also related to a longtime problem in software engineering for code summarization, which is interesting future work.

5.4.4. Synonyms, Correlations, and Concept Maps

Replacing synonyms in queries using the WordNet synonym database has been shown to improve information retrieval results in the other areas of natural language processing, but not in our setting. This is understandable as the API descriptions and queries in our setting is specific to Android development, and many terms are specific to Android, different from the words in the WordNet. For example, the `Intent` class in Android is often used to store data, and may well be treated as a synonym for "data", which is very different from its synonyms in the WordNet. The recommendation results may be improved if we could build domain-specific synonym databases, correlations among related terms, and even a concept map formed by all of the directly or indirectly related words and phrases for Android, which may be done through co-occurrence of words in the Android tutorials and SDK documents, or statistical models and even deep learning of these materials, which can be interesting future work.

5.4.5. Similarity Measurement

We measure similarity between descriptions and queries using a commonly known technique based on *tf-idf*. This is far from perfect, as it has yet considered semantics of the words and is limited by the factors similar to the above (improper descriptions, unmatched synonyms, insufficient correlations, etc.). We could also explore the effects many different variants of similarity measurements, although that may be less an issue if we can construct description databases of high quality and the right semantic level.

5.4.6. Natural Language Parsing

Many factors mentioned above may require advanced natural language parsing techniques. Also, for event callback descriptions and queries, it may be more beneficial if we can identify which phrases in the tutorial and SDK documents are about events (e.g., *when, where, before, after, if*, etc. phrases), and separate the databases for functional APIs and event callbacks. It will be interesting future work to explore this direction too.

5.4.7. Framework Analysis

The implicit control flows and call relations among functional APIs and callbacks in a programming framework are also critical for API recommendation, especially when we want to recommend a set of collaborative APIs together. There are many documents that describe Android framework architecture and how different components in Android work together. Also, the source code of Android SDK contains much rich information on the relations among APIs. It will be even more exciting future work to discover such relations from those documents and source code, by combinations of natural language processing and program analysis techniques, for better recommendation and code usage generation.

5.4.8. Comparison with Other Search Engines

We note that techniques used in other code search and recommendation studies may be adapted for recommending event callbacks although they were designed for functional APIs. In fact, as mentioned in the related work (Section 3), the essential idea of our techniques is similar to many other studies in the sense that it performs a similarity comparison between queries and API description databases; but our study differs in how we generate the description databases from what data. In particular, we notice that it is easy for developers to simply use Google Search scoped to the

`https://developer.android.com/` website to look for potential answers for their queries. However, those search results contain whole web pages; even if a web page in the result list contains the needed APIs and callbacks for developers, it can be too long for developers to read through and identify the APIs needed for programming (although the web pages are good for learning when new developers want to understand the semantics of the APIs better).

That said, it remains an interesting study for us to compare the performance of many different search engines adapted for event callbacks, and even better to incorporate the advantages of various techniques together to improve code recommendation results.

5.4.9. Evaluation Metrics

We have used Precsion@N, Mean Average Precision, and Recall@N to measure the effectiveness of our tool. The metric calculations rely on identifying all relevant answers for a query question, but we can only estimate them based on answers from StackOverflow. Those answers may be incorrect or incomplete answers due to StackOverflow users of different backgrounds and expertises and the limitations of code parsing mentioned in Section 5.4.2, which can affect the construct validity of the metrics. Other metrics [80], such as fall-out, F-score, discounted cumulative gain, may also be used, but may face similar issues too. More online metrics involving interactions with human participants in future work, such as click-through rate, session success rate may be able to show the practical utility of our tool better.

5.4.10. Evaluation with Human Participants

We have not employed human participants (i.e., actual application developers) to validate our recommendation engine. On one hand, we considered that the StackOverflow posts were generated by human and would represent actual questions faced by real users, and that it would be more objective and quantified to measure recommendation results with the posts than with human participants, as the posts are reproducible and there are indicators (e.g., up-votes) for the correctness of the answers. On the other hand, measuring recommendations against the posts is non-trivial either due to the threat to validity of the evaluation metrics (Section 5.4.9), and evaluation with human participants would indeed compensate the threat.

We have added Google Analytics[10] into the LibraryGuru website [13] to

[10] `https://www.google.com/analytics/`

anonymously track query inputs from users and their interactions with the recommendation results (e.g., click-through rate, viewing time with various result pages). We plan in the future to utilize the website to involve more actual users to help in evaluating and improving our recommendation engine.

5.4.11. Generalizability

Although we only demonstrated callback recommendation in the context of Android application development where Java is the main programming language, we expect our idea and techniques (for discovering correlations among functional APIs and callbacks and textual descriptions, generating API descriptions based on code parsing and natural language processing, and matching functional queries based on some similarity measurements) are general enough for other event-driven programming frameworks and languages. We leave a larger scale evaluation of the idea and the techniques for future.

6. Conclusion & Future Work

This paper identifies the need of recommending event callbacks in event-driven programming frameworks and environments, in particular, Android application development. In comparison with functional APIs, our study also shows that callbacks can be more flexible to contain arbitrary code and their documents are often too low level and generic to be directly useful for matching functional queries. This implies that callback recommendations can be a more challenging information retrieval problem than functional API recommendations. We propose an approach for an API recommendation engine that can recommend both (1) functional APIs that can be invoked by developers for implementing certain functionality, and (2) suitable callbacks that can be overridden by the developers to contain their functional code. This approach is based on the essential idea that the right "<Description, API>" relations can be generated from the Android development tutorials and SDK documents at the right semantic level for the functionality requirements that are often faced by developers, and then the relation databases can be used to search for the right APIs for a given functionality query according to certain similarity measurements. Our prototype implementation, named LibraryGuru, has been evaluated on about 1,500 questions related to Android programming collected from StackOverflow, and the results show that our API recommendation engine can produce relevant recommendations with promising precisions.

Although our study demonstrates that recommendation of event callbacks is needed and can be done for developers, the design, implementation, evaluation of our API recommendation engine has many limitations so far, such as limited data sources, straightforward API description generation, simple similarity measurements, relatively small evaluation, etc. In the near future, we aim to improve our API recommendation engine by enhancing description generation and query matching with more accurate domain-specific synonyms, correlations, and concept maps, and more semantic-aware code and language parsing and analysis. Particularly for callbacks, we believe that providing richer documents with more usage scenarios (by the framework and API developers) in addition to SDK documents, utilizing larger sets of open-source applications (by a large community of application developers) and contexts where questions are raised (by individual API client application developers) can be useful in improving the recommendation results.

Acknowledgements

This work was supported by the 973 Program in China (Grant No. 2015CB352203), the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 Grant (Fund No. C220/MSS16C003), and the National Nature Science Foundation of China (Grant No. 61572312). We greatly appreciate the supports from our funding agencies and useful feedback from anonymous reviewers.

References

[1] F. Thung, S. Wang, D. Lo, J. L. Lawall, Automatic recommendation of API methods from feature requests, in: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 290–300. doi:10.1109/ASE.2013.6693088.

[2] H. Yu, W. Song, T. Mine, APIBook: An effective approach for finding APIs, in: Proceedings of the 8th Asia-Pacific Symposium on Internetware, ACM, 2016, pp. 45–53. doi:10.1145/2993717.2993727.

[3] M. M. Rahman, C. K. Roy, D. Lo, RACK: Automatic API recommendation using crowdsourced knowledge, in: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, 2016, pp. 349–359. doi:10.1109/SANER.2016.80.

[4] X. Gu, H. Zhang, D. Zhang, S. Kim, Deep api learning, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, pp. 631–642. doi:10.1145/2950290.2950334.

[5] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei, MAPO: Mining and recommending api usage patterns, in: Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP), 2009, pp. 318–343. doi:10.1007/978-3-642-03013-0_15.

[6] R. P. L. Buse, W. Weimer, Synthesizing API usage examples, in: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 782–792. doi:10.1109/ICSE.2012.6227140.

[7] T. V. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, T. N. Nguyen, T2API: Synthesizing API code usage templates from english texts with statistical translation, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, pp. 1013–1017. doi:10.1145/2950290.2983931.

[8] H. A. Nguyen, R. Dyer, T. N. Nguyen, H. Rajan, Mining preconditions of apis in large-scale code corpus, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 166–177. doi:10.1145/2635868.2635924.

[9] E. Moritz, M. L. Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, M. Gethers, ExPort: Detecting and visualizing API usages in large source code repositories, in: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 646–651. doi:10.1109/ASE.2013.6693127.

[10] M. A. Saied, H. Abdeen, O. Benomar, H. Sahraoui, Could we infer unordered API usage patterns only using the library source code?, in: Proceedings of the IEEE 23rd International Conference on Program Comprehension (ICPC), 2015, pp. 71–81. doi:10.1109/ICPC.2015.16.

[11] H. Niu, I. Keivanloo, Y. Zou, API usage pattern recommendation for software development, J. Syst. Softw. 129 (C) (2017) 127–139. doi:10.1016/j.jss.2016.07.026.

[12] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, P. Ou, Automatic parameter recommendation for practical API usage, in: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 826–836. doi:10.1109/ICSE.2012.6227136.

[13] W. Yuan, H. H. Nguyen, L. Jiang, Y. Chen, LibraryGuru: API recommendation for android developers, in: 40th International Conference on Software Engineering (ICSE) Posters, 2018.
URL `http://libraryguru.info`

[14] StackOverflow, All newest questions, `https://stackoverflow.com/questions` (August 2017).

[15] StackOverflow, All newest 'android' questions tagged, `https://stackoverflow.com/questions/tagged/android` (August 2017).

[16] M. P. Robillard, W. Maalej, R. J. Walker, T. Zimmermann (Eds.), Recommendation Systems in Software Engineering, Springer-Verlag Berlin Heidelberg, 2014. doi:10.1007/978-3-642-45135-5.

[17] J. Wu, L. Shen, W. Guo, W. Zhao, How is code recommendation applied in android development: A qualitative review, in: International Conference on Software Analysis, Testing and Evolution (SATE), 2016, pp. 30–35. doi:10.1109/SATE.2016.12.

[18] J. Wu, L. Shen, W. Guo, W. Zhao, Code recommendation for android development: How does it work and what can be improved?, Science China Information Sciences 60 (9) (2017) 092111. doi:10.1007/s11432-017-9058-0.

[19] F. Thung, R. J. Oentaryo, D. Lo, Y. Tian, WebAPIRec: Recommending web APIs to software projects via personalized ranking, IEEE Trans. Emerging Topics in Comput. Intellig. 1 (3) (2017) 145–156. doi:10.1109/TETCI.2017.2699222.

[20] F. Xie, J. Liu, M. Tang, D. Zhou, B. Cao, M. Shi, Multi-relation based manifold ranking algorithm for api recommendation, in: G. Wang, Y. Han, G. Martínez Pérez (Eds.), Advances in Services Computing, 2016, pp. 15–32. doi:10.1007/978-3-319-49178-3_2.

[21] L. F. H. Ramírez, H. A. X. Costa, M. T. d. O. Valente, A. P. Freire, P. A. Parreira Júnior, API recommendation system in software engineering, Master's thesis, Universidade Federal de Lavras (2016).
URL `http://repositorio.ufla.br/jspui/handle/1/11145`

[22] W. Shi, X. Sun, B. Li, Y. Duan, X. Liu, Using feature-interface graph for automatic interface recommendation: A case study, in: Third International Conference on Advanced Cloud and Big Data, 2015, pp. 296–303. doi:10.1109/CBD.2015.55.

[23] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, C. Fu, Portfolio: Searching for relevant functions and their usages in millions of lines of code, ACM Trans. Softw. Eng. Methodol. 22 (4) (2013) 37:1–37:30. doi:10.1145/2522920.2522930.

[24] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, C. Fu, Portfolio: finding relevant functions and their usage, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE), 2011, pp. 111–120. doi:10.1145/1985793.1985809.

[25] W.-K. Chan, H. Cheng, D. Lo, Searching connected api subgraph via text phrases, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE), 2012, pp. 10:1–10:11. doi:10.1145/2393596.2393606.

[26] F. Thung, D. Lo, J. L. Lawall, Automated library recommendation, in: 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 182–191. doi:10.1109/WCRE.2013.6671293.

[27] C. Chen, Z. Xing, SimilarTech: Automatically recommend analogical libraries across different programming languages, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 834–839. doi:10.1145/2970276.2970290.

[28] C. McMillan, M. Grechanik, D. Poshyvanyk, Detecting similar software applications, in: 34th International Conference on Software Engineering (ICSE), 2012, pp. 364–374. doi:10.1109/ICSE.2012.6227178.

[29] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, Q. Xie, Exemplar: A source code search engine for finding highly relevant applications, IEEE Trans. Software Eng. 38 (5) (2012) 1069–1087. doi:10.1109/TSE.2011.84.

[30] M. P. Robillard, Y. B. Chhetri, Recommending reference api documentation, Empirical Softw. Engg. 20 (6) (2015) 1558–1586. doi:10.1007/s10664-014-9323-y.

[31] L. Guerrouj, D. Bourque, P. C. Rigby, Leveraging informal documentation to summarize classes and methods in context, in: IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), Vol. 2, 2015, pp. 639–642. doi:10.1109/ICSE.2015.212.

[32] N. Nazar, Y. Hu, H. Jiang, Summarizing software artifacts: A literature review, Journal of Computer Science and Technology 31 (5) (2016) 883–909. doi:10.1007/s11390-016-1671-1.

[33] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, T. Ratchford, Automated api property inference techniques, IEEE Trans. Softw. Eng. 39 (5) (2013) 613–637. doi:10.1109/TSE.2012.63.

[34] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, E. Wong, On-demand developer documentation, in: IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 479–483. doi:10.1109/ICSME.2017.17.

[35] M. Kechagia, D. Mitropoulos, D. Spinellis, Charting the api minefield using software telemetry data, Empirical Softw. Engg. 20 (6) (2015) 1785–1830. doi:10.1007/s10664-014-9343-7.

[36] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, D. Zhang, Mining succinct and high-coverage API usage patterns from source code, in: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), 2013, pp. 319–328. doi:10.1109/MSR.2013.6624045.

[37] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, W. Hu, Bing developer assistant: Improving developer productivity by recommending sample code, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, pp. 956–961. doi:10.1145/2950290.2983955.

[38] Z. Zhu, Y. Zou, Y. Jin, B. Xie, Generating api-usage example for project developers, in: Proceedings of the 5th Asia-Pacific Symposium on Internetware, 2013, pp. 34:1–34:4. doi:10.1145/2532443.2532470.

[39] S. Azad, P. C. Rigby, L. Guerrouj, Generating api call rules from version history and stack overflow posts, ACM Trans. Softw. Eng. Methodol. (TOSEM) 25 (4) (2017) 29:1–29:22. doi:10.1145/2990497.

[40] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, How can I use this method?, in: Proceedings of the 37th International Conference on Software Engineering (ICSE), 2015, pp. 880–890. doi:10.1109/ICSE.2015.98.

[41] T. T. Nguyen, H. V. Pham, P. M. Vu, T. T. Nguyen, Learning api usages from bytecode: A statistical approach, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 416–427. doi:10.1145/2884781.2884873.

[42] P. Rodeghero, C. McMillan, A. Shirey, API usage in descriptions of source code functionality, in: 1st IEEE/ACM International Workshop on API Usage and Evolution, WAPI@ICSE 2017, 2017, pp. 3–6. doi:10.1109/WAPI.2017.3.

[43] C. McMillan, D. Poshyvanyk, M. Grechanik, Recommending source code examples via API call usages and documentation, in: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering (RSSE), 2010, pp. 21–25. doi:10.1145/1808920.1808925.

[44] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, T. N. Nguyen, Statistical learning approach for mining api usage mappings for code migration, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE), 2014, pp. 457–468. doi:10.1145/2642937.2643010.

[45] J. Fowkes, C. Sutton, Parameter-free probabilistic api mining across github, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, pp. 254–265. doi:10.1145/2950290.2950319.

[46] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, D. Dig, API code recommendation using statistical learning from fine-grained changes, in: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016, pp. 511–522. doi:10.1145/2950290.2950333.

[47] M. Bruch, T. Schäfer, M. Mezini, On evaluating recommender systems for API usages, in: Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE), 2008, pp. 16–20. doi:10.1145/1454247.1454254.

[48] E. Duala-Ekoko, M. P. Robillard, Using structure-based recommendations to facilitate discoverability in APIs, in: Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP), 2011, pp. 79–104. doi:10.1007/978-3-642-22655-7_5.

[49] Y. Ye, Y. Yamamoto, K. Nakakoji, Y. Nishinaka, M. Asada, Searching the library and asking the peers: Learning to use java apis on demand, in: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ), 2007, pp. 41–50. doi:10.1145/1294325.1294332.

[50] K. Yessenov, I. Kuraj, A. Solar-Lezama, Demomatch: Api discovery from demonstrations, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2017, pp. 64–78. doi:10.1145/3062341.3062386.

[51] M. Ichinco, W. Y. Hnin, C. L. Kelleher, Suggesting api usage to novice programmers with the example guru, in: Proceedings of the CHI Conference on Human Factors in Computing Systems, 2017, pp. 1105–1117. doi:10.1145/3025453.3025827.

[52] P. Roos, Fast and precise statistical code completion, in: Proceedings of the 37th International Conference on Software Engineering (ICSE), 2015, pp. 757–759. doi:10.1109/ICSE.2015.240.

[53] D. Mandelin, L. Xu, R. Bodík, D. Kimelman, Jungloid mining: Helping to navigate the api jungle, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2005, pp. 48–61. doi:10.1145/1065010.1065018.

[54] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. M. Al-Kofahi, T. N. Nguyen, Graph-based pattern-oriented, context-sensitive source code completion, in: 34th International Conference on Software Engineering (ICSE), 2012, pp. 69–79. doi:10.1109/ICSE.2012.6227205.

[55] M. Bruch, M. Monperrus, M. Mezini, Learning from examples to improve code completion systems, in: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE), 2009, pp. 213–222. doi:10.1145/1595696.1595728.

[56] S. Chatterjee, S. Juvekar, K. Sen, SNIFF: A search engine for java using free-form queries, in: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE): Held As Part of the Joint European Conferences on Theory and Practice of Software (ETAPS), 2009, pp. 385–400. doi:10.1007/978-3-642-00593-0_26.

[57] M. Raghothaman, Y. Wei, Y. Hamadi, SWIM: Synthesizing what i mean: Code search and idiomatic snippet synthesis, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 357–367. doi:10.1145/2884781.2884808.

[58] E. C. Campos, M. Monperrus, M. A. Maia, Searching stack overflow for API-usage-related bug fixes using snippet-based queries, in: Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON), 2016, pp. 232–242.
URL http://dl.acm.org/citation.cfm?id=3049877.3049902

[59] L. Ponzanelli, Holistic recommender systems for software engineering, in: 36th International Conference on Software Engineering (ICSE), 2014, pp. 686–689. doi:10.1145/2591062.2591081.

[60] L. Richardson, Beautiful soup, https://www.crummy.com/software/BeautifulSoup/ (Aug. 2017).

[61] I. B. Stefan Behnel, Martijn Faassen, lxml - xml and html with python, http://lxml.de/.

[62] Eclipse Foundation, Eclipse Java Development Tools (JDT), http://www.eclipse.org/jdt/ (Aug. 2017).

[63] Z. S. Harris, Distributional structure, Word 10 (2-3) (1954) 146–162. doi:10.1007/978-94-009-8467-7_1.

[64] E. Loper, S. Bird, NLTK: The natural language toolkit, in: Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for

Teaching Natural Language Processing and Computational Linguistics (ETMTNLP), 2002, pp. 63–70. doi:10.3115/1118108.1118117.

[65] G. Salton, A. Wong, C.-S. Yang, A vector space model for automatic indexing, Communications of the ACM 18 (11) (1975) 613–620. doi:10.1145/361219.361220.

[66] H. P. Luhn, A statistical approach to mechanized encoding and searching of literary information, IBM Journal of Research and Development 1 (4) (1957) 309–317. doi:10.1147/rd.14.0309.

[67] K. Sparck Jones, A statistical interpretation of term specificity and its application in retrieval, Journal of Documentation 28 (1) (1972) 11–21. doi:10.1108/eb026526.

[68] R. Subhashini, V. J. S. Kumar, Evaluating the performance of similarity measures used in document clustering and information retrieval, in: First International Conference on Integrated Intelligent Computing, 2010, pp. 27–31. doi:10.1109/ICIIC.2010.42.

[69] S. I. Hajeer, Comparison on the effectiveness of different statistical similarity measures, International Journal of Computer Applications (IJCA) 53 (8) (2012) 14–19. doi:10.5120/8440-2224.

[70] A. Heidarian, M. J. Dinneen, A hybrid geometric approach for measuring similarity level among documents and document clustering, in: IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService), 2016, pp. 142–151. doi:10.1109/BigDataService.2016.14.

[71] G. A. Miller, WordNet: A lexical database for english, Communication of ACM 38 (11) (1995) 39–41. doi:10.1145/219717.219748.

[72] A. Turpin, F. Scholer, User performance versus precision measures for simple search tasks, in: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 2006, pp. 11–18. doi:10.1145/1148170.1148176.

[73] D. Klein, C. D. Manning, Accurate unlexicalized parsing, in: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1, 2003, pp. 423–430. doi:10.3115/1075096.1075150.

[74] T. Parr, S. Harwell, K. Fisher, Adaptive LL(*) parsing: The power of dynamic analysis, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA), 2014, pp. 579–598. doi:10.1145/2660193.2660202.

[75] A. Afroozeh, A. Izmaylova, One parser to rule them all, in: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), 2015, pp. 151–170. doi:10.1145/2814228.2814242.

[76] P. Yin, G. Neubig, A syntactic neural model for general-purpose code generation, in: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL), 2017, pp. 440–450. doi:10.18653/v1/P17-1041.

[77] L. Dong, M. Lapata, Coarse-to-fine decoding for neural semantic parsing, in: 56th Annual Meeting of the Association for Computational Linguistics (ACL), 2018, pp. 731–742.
URL `http://aclweb.org/anthology/P18-1068`

[78] M. Rabinovich, M. Stern, D. Klein, Abstract syntax networks for code generation and semantic parsing, in: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL), 2017, pp. 1139–1149. doi:10.18653/v1/P17-1105.

[79] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, J. N. Amaral, Syntax and sensibility: Using language models to detect and correct syntax errors, in: IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 311–322. doi:10.1109/SANER.2018.8330219.

[80] C. D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008, Ch. Chapter 8: Evaluation in information retrieval.
URL `https://nlp.stanford.edu/IR-book/`