Information and Software Technology 138 (2021) 106619

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# *Icon2Code*: Recommending code implementations for Android GUI components

Yanjie Zhao, Li Li *, Xiaoyu Sun, Pei Liu, John Grundy

*Faculty of Information Technology, Monash University, Melbourne, Australia*

ARTICLE INFO

*Keywords:*
Android
App development
Collaborative filtering
Icon implementation
API recommendation

ABSTRACT

**Context:** Event-driven programming plays a crucial role in implementing GUI-based software systems such as Android apps. However, such event-driven code is inherently challenging to design and implement correctly. Despite a significant amount of research to help developers efficiently implement such software, improved approaches are still needed to assist developers in better handling events and associated callback methods.
**Objective:** This work aims at inventing an intelligent recommendation system for helping app developers efficiently and effectively implement Android GUI components.
**Methods:** To achieve the aforementioned objective, we introduce in this work a novel approach called Icon2Code. Given an icon or UI widget provided by designers as input, Icon2Code first searches from a large-scale app database to locate similar icons used in existing popular apps. It then learns from the implementation of these similar apps and leverages a collaborative filtering model to select and recommend the most relevant APIs.
**Results:** Our approach can achieve an 81% success rate when only five recommended APIs are considered, and a 94% success rate if twenty results are considered, based on ten-fold cross-validation with a large-scale dataset containing over 45,000 icons and their code implementations.
**Conclusion:** It is feasible to automatically recommend code implementations for Android GUI components and Icon2Code is useful and effective in helping achieve such an objective.

## 1. Introduction

With over 2.7 billion users worldwide using smartphones, it is no surprise that the mobile app industry is thriving. It is expected that mobile apps will generate $189 billion in revenue by 2020, which is larger than the projected 2020 GDPs of many developed countries, such as Canada and Australia. Android, occupying over 80% of market shares, is undoubtedly the most prominent mobile platform. Currently, around 2.8 million Android apps are available on the official Google Play store for users to download, and this number continually grows year-over-year.

The huge number of available apps provides users with a wide range of opportunities to choose apps to install. However, it also forces developers to develop and update their apps in a timely manner as competition is some of the fiercest in the world [1]. As a consequence, developers often adopt very short release cycles to keep their apps competitive. This includes reasons such as to cope with new mobile devices or OS versions, resolve negative user feedback, and rapidly introduce new features. Nevertheless, it is non-trivial to keep releasing apps in such short cycles, and developers are often under high-pressure to fix vulnerabilities, bugs, and compatibility issues [2–5] and to cope with learning new development methodologies, libraries, and state-of-the-art technologies [6–8]. To assist them the software engineering community has proposed various approaches to ease developers' work in keeping their apps up-to-date [9,10]. For example, automated API usage recommendation approaches to strengthen the development of mobile apps [11,12], as well as other software systems [13–15]. These approaches have been experimentally demonstrated to be useful and effective in helping developers completing their implementation tasks.

Unfortunately, to the best of our knowledge, none of the existing approaches have been proposed to support code implementation for Android apps' GUI component event handlers. GUI is a ubiquitous feature for all mobile apps, which are event-centric programs driven by rich graphical user interface interactions with users. One of the major complexities of implementing mobile app GUIs is managing the complicated and intertwined callback events from user interaction. This often takes a major amount of coding and debugging efforts [12,16]. Fortunately, functional APIs are capable of helping developers implement the functionalities in callback methods. Indeed, as empirically

* Corresponding author.
*E-mail address:* Li.Li@monash.edu (L. Li).

https://doi.org/10.1016/j.infsof.2021.106619
Received 29 October 2020; Received in revised form 28 February 2021; Accepted 7 May 2021
Available online 27 May 2021

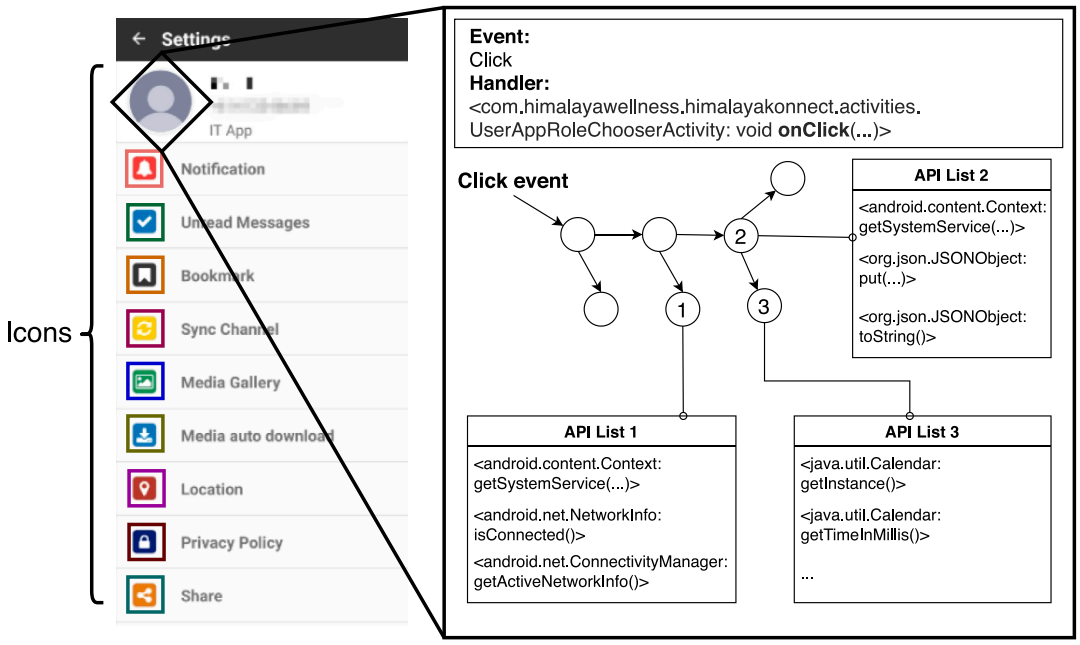



**Fig. 1.** Example of icon-bound GUI components and icon-associated event handler callback method. The callback method can access various methods and among which each of them can further access different Android and third-party library APIs.

disclosed by Gao et al. [17], API recommendation is useful for the development of Android app features, which could be strongly bound to certain GUI icons (e.g., a camera icon could indicate a feature of taking photos). Nevertheless, it is still a time-consuming task to identify and correctly use the appropriate functional APIs to fulfill the callback methods for implementing feature requirements [12,18].

To help developers efficiently and effectively implement these callback methods of GUI components, we propose a prototype tool called *Icon2Code* to learn similar implementations from existing Android apps. *Icon2Code* is based on the premise that similar GUI components are often designed to have similar app behaviors e.g., a heart icon for *like*, and this results in similar code implementations. *Icon2Code* aims to capture such common implementations to assist app developers in implementing interactions driven by GUI components. *Icon2Code* first leverages a static analysis module to parse the code of existing apps and build mappings from GUI components to their associated callback methods. Then for each callback method, *Icon2Code* extracts its call graph and summarizes all of its accessed APIs, including those of third-party libraries. Finally *Icon2Code* leverages a collaborative filtering algorithm to build a recommendation system. This takes as input a GUI component (i.e., an icon) and outputs a list of code implementation bundles learned from such apps sharing similar GUI components. The main contributions of this work include:

- *Icon2Code*, a prototype tool that takes as input an icon or text describing its purpose and outputs a ranked list of APIs recommended for implementing the icon-associated callback method, or event handler;
- a large-scale training database containing mappings from icons to their code implementations;
- ten-fold cross-validation reveals that our *Icon2Code* approach is useful and effective in recommending code implementation for Android GUI components, achieving 81% success rate when only the top five recommended APIs are considered, and a 94% success rate if twenty results are considered.

Section 2 presents a motivating example for the need for *Icon2Code*. Section 3 presents the approach and key workings of *Icon2Code*, and Section 4 describes its evaluation. We discuss the threats to validity and future work in Section 5. The closely related works are detailed in Section 6, followed by the summarization in Section 7.

## 2. Motivation

As argued by Chen et al. [19], developing the GUI of an app involves two separate activities: (1) Design of the GUI and (2) Implementation of the GUI. The former activity is often done by professional designers as creating an intuitive and pleasant user interface is crucial for an app's success in the highly competitive market. The latter usually involves the implementation of the GUI interface itself e.g. GUI widgets details, layout, constraints, and handling of user interactions such as what happens when a button is clicked. State-of-the-art approaches that have attempted to generate GUI interfaces automatically [19] have proposed a neural machine translator to translate GUI design images to GUI skeletons. They have not attempted to help developers quickly implement user interactions code for the GUI components in the user interfaces.

Listing 1: Examples of API usages in the method (Node 2 in Fig. 1) reached by the click event. It is worth mentioning that both Android APIs and third-party library APIs are used in this method.

```
1 //Node 2
2 TelephonyManager telephonyManager = (
      TelephonyManager) this.la.
      getSystemService("phone");
3 String str = telephonyManager.
      getNetworkCountryIso().toUpperCase();
4 JSONObject jSONObject = new JSONObject();
5 jSONObject.put("isoccode", str);
6 new b(..., jSONObject.toString(), ...);
```

Consider Fig. 1 as an example. This is a typical GUI page extracted from an Android app *com.himalayawellness.hi malayakonnect*. As highlighted, this GUI page contains various GUI components referred to as **icons** through this paper. Each of these icon GUI components must take user inputs, such as a click or other interaction, and respond accordingly. For example, when users click the *Profile* icon (top left), a new page will be switched to. This should allow users to configure their profile data for the app. Such behavior changes driven by user inputs are usually done by so-called **callback methods**. Ideally, each icon should be associated with at least one callback method. The implementations of these callback methods are often quite complicated,

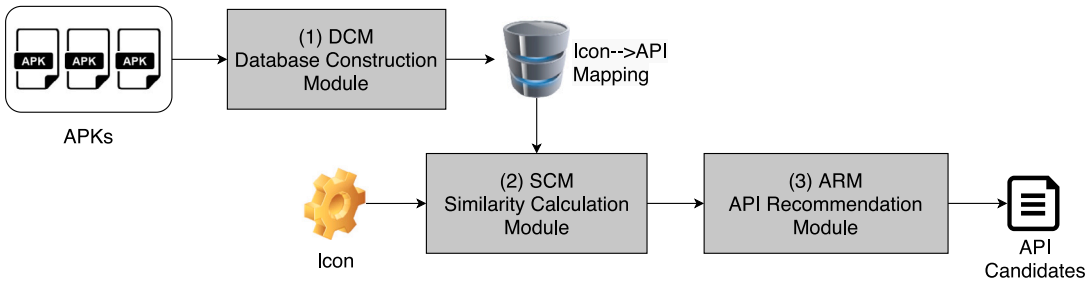



**Fig. 2.** The architecture of *Icon2Code*. Each icon involved in this work includes an icon image file and its description text given by app developers.

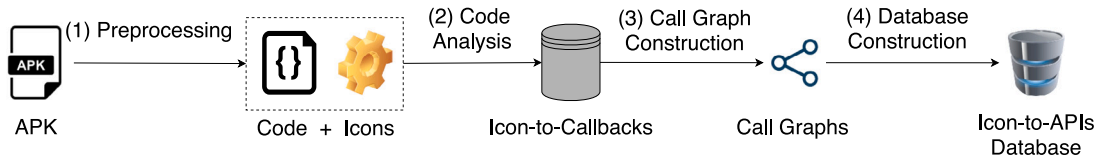



**Fig. 3.** The working process of the first module DCM. Each icon involved in this work includes an icon image file and its description text given by app developers.

involving various method calls accessing multiple Android APIs and possibly third-party libraries. As such an example, let us consider the simplified code snippet, shown in Listing 1, extracted from one of the methods (i.e., node 2) in the call chain triggered by the *onClick* callback method (i.e., once the icon is clicked), this single method is involved in at least three APIs including both Android and third-party library APIs. A single callback can access many such methods. Furthermore, a single GUI page can contain dozens of icons, i.e., callback methods, making it even harder to correctly implement and debug. The event-driven nature of such user interfaces is well known to be challenging to code [20,21]. However, to the best of our knowledge, no existing approaches help developers implement those complicated callback methods associated with Android GUI icons.

## 3. *Icon2Code*

Fig. 2 presents an overview of our *Icon2Code* prototype tool. Its three key modules are (1) Database Construction Module (DCM), (2) Similarity Calculation Module (SCM), and (3) API Recommendation Module (ARM).

### 3.1. DCM: Database Construction Module

The key objective of this work is to recommend code implementations for GUI icons. We achieve this purpose by learning from the code implementations of existing apps that have used similar icons in their GUIs. To this end, the first module of *Icon2Code* aims to pre-analyze a broad set of Android apps to construct a database mapping icon to its code implementation, i.e. Android code, JDK, and Android APIs, as well as third-party library APIs (i.e., user-defined APIs or methods are ignored). Fig. 3 shows the working process of this module.

#### 3.1.1. Preprocessing

The first step preprocesses Android APKs to extract useful information (such as the icons) to prepare for further analysis. Android application package (APK) is the file format used to distribute and install applications on Android. In APKs there are two primary forms of icons: the *vector* icon defined by an XML file, and the *image* icon provided as images files (such as PNG files). Since the former does not directly come as images, we skip this form in this work and only consider the latter. When disassembling Android APKs, we also parse the manifest file to extract the targeted SDK versions, by extracting the values of *minSdkVersion* and *targetSdkVersion*). This information will be used when recommending APIs for Android apps under development, i.e., the recommended APIs should align with the targeted SDK versions of the target app.

**Table 1**
A set of attributes responsible for binding icons to GUI components.

| Attribute | Explanation |
|---|---|
| android:src | Set a drawable as the content of the view (e.g., ImageView). |
| android:background | Set a drawable as the background of the view. |
| android:drawableRight | Set a drawable to the right of the text. |
| android:drawableTop | Set a drawable on top of the text. |
| android:drawableLeft | Set a drawable to the left of the text. |
| android:drawableBottom | Set a drawable below the text. |
| android:drawableEnd | Set a drawable at the end of the text. |
| android:drawableStart | Set a drawable at the start of the text. |

Listing 2: Examples of using XML attributes to bind icons with GUI components.

```
1  //Example 1: ImageView, android:src
2  <ImageView android:src="@drawable/
       next_btn"
3   android:id="@+id/nextBtn"
4   android:contentDescription="@string/
        next_button_content_desc"
5   android:onClick="onClick"/>
6
7  //Example 2: Button, android:background
8  <Button android:background="@drawable/
       button_cancel"
9   android:id="@+id/cancel_btn"
10  android:text="@string/cancel"/>
```

#### 3.1.2. Code analysis

This step statically analyzes program code in Android APKs to establish a mapping from icon-bound GUI components and their corresponding callback methods that respond to user events. To achieve this, we analyze how icons are bound to GUI components on an app UI page. We observe that icons are generally bound through XML attributes in the apps' layout configuration files. For example, the source of an ImageView (i.e., android:src), the background image of a Button (i.e., android:back-ground). Listing 2 shows two such examples on lines 2 and 8. Table 1 summarizes the list of attributes we have considered in this work, and this list has already been leveraged by other Android analysis techniques [22].

Listing 3: An example of dynamically defining an icon's event handler (i.e., callback method) through program code.

```
1  public class MusicWallpaper extends
       Activity implements View.
       OnClickListener {
2   public void onCreate(Bundle bundle) {
3    setContentView(R.layout.layoutlagu);
4    ImageView v = (ImageView) findViewById(
         R.id.nextBtn);
5    //Binding callback method to the icon
6    v.setOnClickListener(this);
7   }
8   @Override
9   public void onClick(View view) {
10   //This is the callback method
11   ...
12 }}
```

After locating GUI components, this step's second task is to infer their associated callback methods. As shown in Listing 3, it is non-trivial to achieve this, as callback methods can be associated with GUI components in two different ways. Like the bindings between GUI components and icon files, callback methods can be specified through XML attributes. Listing 2 demonstrates such an example. The attribute *android:onClick* (line 5) specifies the callback method (also here named as *onClick*) that is triggered if the image view is clicked. This type of binding can be easily resolved by parsing the layout configuration files. On the other hand, instead of statically defining the callback methods, Android app developers can make the binding dynamically in program code. Using the same *onClick* callback method, instead of using the XML attribute (line 5 in Listing 2), developers can leverage code as shown in Listing 3 to achieve the same purpose. This type of binding is much more challenging to identify. Fortunately in most cases the callback methods are added following the creation of a GUI component (e.g., *findViewById()* at line 4). By statically connecting those statements, one can eventually make a mapping from icons to their dynamically defined callback methods.

#### 3.1.3. Call graph construction

Based on the mapping from icons to callback methods, we then harvest the set of APIs accessed by the aforementioned callback methods and consequently build a mapping from icons to their corresponding set of APIs, called when the icons receive user inputs such as being clicked. Unfortunately, as shown by our motivating example, it is not straightforward to achieve this. A given callback method may access a set of other methods, and each can invoke a set of APIs. We resort to static code analysis to construct call graphs to ease the extraction of APIs. For each callback method that we consider as an entry point, we construct a call graph for it with nodes representing methods and edges representing method invocations.

#### 3.1.4. Database construction

For each icon identified previously, *Icon2Code* traverses its call graph and extracts all Android and third-party APIs accessed by methods in the graph. It then puts this mapping of icon to its associated APIs into a database. We consider this to be the ground truth to support learning of API recommendations. Ideally, the more apps considered for training, the more comprehensive the database will be, and subsequently, the more reliable the API recommendation approach could be. Additionally, *Icon2Code* further records completed API usage examples, like the code snippet shown in Listing 1, into the database. This allows Icon2Code to recommend further API usage examples. We believe this will be useful and helpful for developers to master the recommended APIs more quickly. As noted previously, building event-driven interfaces is challenging and this helps them to more easily reuse appropriate icon-related code, especially for complex screens.

### 3.2. SCM: Similarity Calculation Module

Given an icon as input, SCM will locate similar icons from the pre-built *icon* → *APIs* database. Given an icon and a similarity threshold, the pre-trained database may return thousands of similar icons. We introduce a configurable parameter $m$ to control the number of most similar icons for analysis.

Since icons in Android apps can be associated with alternative text describing the icon's purpose, such text is also leveraged to identify similar icons. We have identified three main sources that provide alternative text to icon-bound GUI components: (1) **S1**: The icon's reference name (e.g., through *android:src*), which often describes the function of the icon; (2) **S2**: The id name of the view (e.g., through *android:id*) where the icon bound to, which often describes the function of the view hosting the icon; and (3) **S3**: The alternative text defined via *android:contentDescription* or *android:text* XML attributes, designed to describe the function of the view. All three alternative texts are supposed to specify the purpose of the view and should be similar to some extent. Take Listing 2 as an example — alternative texts {S1, S2, S3} of the two examples are {next_btn, nextBtn, next_button_content_desc}, {button_cancel, cancel_btn, cancel}. These are indeed similar to each other in a set of alternative texts and align with the purpose of the view and icon. In addition to a direct comparison between icon images, we also use alternate text similarity to find the top-$m$ most similar icons.

**Image similarity calculation.** We rely on straightforward approaches to measure the similarity of icons. Such approaches, although easy to implement, may not be reliable in practice. We introduce three algorithms to calculate similarities of images, Oriented FAST and Rotated BRIEF (ORB) algorithm [23], Locality Sensitive Hashing (LSH) algorithm [24], and a traditional Histogram algorithm [25]. Given two images $p$ and $q$, their similarity is calculated by Formula (1), where the fusion similarity threshold is defined as 0.85. If the maximum value of the similarity calculated by the three algorithms is greater than or equal to 0.85, the maximum value will be taken as the final similarity. Otherwise, the minimum value of the similarity calculated by the three algorithms will be considered as the ultimate similarity of the fusion algorithm. Through this hybrid image similarity calculation method, the random error caused by a single method can be significantly reduced.

$$\begin{aligned} Let\ Max_s &= max(ORB(p,q), LSH(p,q), Histogram(p,q)),\\ Let\ Min_s &= min(ORB(p,q), LSH(p,q), Histogram(p,q)),\\ Sim_{image}(p,q) &= \begin{cases} Max_s, & Max_s \geq 0.85 \\ Min_s, & Max_s < 0.85 \end{cases} \end{aligned} \tag{1}$$

**Text similarity calculation.** To ascertain the similarity of two text strings, *edit distance* is a widely-used method that computes the minimum number of edit operations required to transform one text into the other [26]. *Levenshtein distance* is such a type of commonly used edit distance [27], upon which the semantic similarity of two texts can be represented using the *Levenshtein ratio* [28]. Given $a$ and $b$ as two texts, their *Levenshtein ratio* score can hence be calculated following Formula (2). A perfect match will achieve a Levenshtein score of 1, while an entirely dissimilar case will result in a score of 0. Given two icons $p$ and $q$, their alternate text similarity is calculated by Formula (3), where $p'$ and $q'$ are the alternative text of $p$ and $q$, and $w_1, w_2, w_3$ are the weights of each type of alternative text, i.e., $S1, S2, S3$, separately.

$$LevenshteinRatio(a,b) = 1 - \frac{LevenshteinDistance(a,b)}{|a| + |b|} \tag{2}$$

$$\begin{aligned} Sim_{text}(p',q') = \quad & w_1 \times LevenshteinRatio_{S1}(p',q') \quad + \\ & w_2 \times LevenshteinRatio_{S2}(p',q') \quad + \\ & w_3 \times LevenshteinRatio_{S3}(p',q') \end{aligned} \tag{3}$$

We aggregate these two similarity calculation algorithms to compute the overall similarity of two icons via Formula (4), where $\alpha$ and $\beta$ represent the weights of $Sim_{image}$ and $Sim_{text}$, respectively.

$$Sim(p,q) = \alpha \times Sim_{image}(p,q) + \beta \times Sim_{text}(p',q') \tag{4}$$

**Table 2**
An example of the encoding matrix.

| | $api_1$ | $api_2$ | ⋯ | $api_k$ |
|---|---|---|---|---|
| $i_1$ | 1 | 0 | 1 | 0 |
| $i_2$ | 1 | 1 | 0 | 1 |
| ⋯ | 1 | 1 | 1 | 0 |
| $i_m$ | 0 | 1 | 1 | 0 |
| $i_{edit}$ | −1 | −1 | −1 | −1 |

### 3.3. ARM: API recommendation module

ARM learns from a set of code implementations to recommend APIs for the input icon that is under development, i.e. developers want to implement its corresponding callback method(s). *Icon2Code* leverages collaborative filtering to recommend API usages. Schafer et al. [29] present collaborative filtering (CF) as a process of filtering or evaluating items through the opinions of other people. The approach has often been used to recommend items for users to purchase based on past shopping records or the records of other users with similar purchasing behaviors. In *Icon2Code*, an icon plays the role of a *user*, while each API plays the role of an *item*. The goal of ARM is hence to recommend users (icons) a list of items (APIs) to purchase (to access).

Based on the $m$ most similar icons returned by SCM module, *Icon2Code* first determines the number of APIs ($k$) accessed by the associated callback methods of the selected icons and models them into a $(m+1) * k$ matrix. Table 2 illustrates such an example. Icons – selected ones $i_1 \rightarrow i_m$ plus the one under development $i_{edit}$ – are represented as rows while APIs are represented as columns. For the selected $m$ icons, each of their cells in the matrix is set to either true (1) or false (0), representing whether the icon-related callback methods have accessed the corresponding API or not. For example, cell $(i_2, api_k)$ is set to be 1, indicating that callbacks of icon $i_2$ has accessed $api_k$. For the icon under development (i.e., $i_{edit}$ in the last row), all of its cells will be set to unknown (−1). The goal of this module is hence switched to predict possible values for those unknown cells. The cells received higher values – or the corresponding APIs – will then be recommended for app developers to complete the development of the icon-associated callback method.

The probability of recommending a given API $api$ to $i_{edit}$ can be calculated via Formula (5) [29], where $neighbors(i_{edit})$ is the set of the $m$ most similar icons, $sim(i_{edit}, i)$ is defined by Formula (4), and $r_{i_{edit}}^{-}$ and $\bar{r_i}$ are the mean ratings of $i_{edit}$ and $i$, respectively. In our implementation, $\bar{r_i}$ and $r_{i,api}$ are obtained from the encoding matrix. For example, for the encoding matrix shown in Table 2, we could calculate $\bar{r_i}$ by measuring the average rating of the cells in the row corresponding to $i$. For $r_{i_{edit}}^{-}$, we set its value to 0.8 following the general practice of the state-of-the-art [30].

$$p_{i_{edit},api} = r_{i_{edit}}^{-} + \frac{\sum_{i \in neighbors(i_{edit})} (r_{i,api} - \bar{r_i}) \cdot sim(i_{edit}, i)}{\sum_{i \in neighbors(i_{edit})} sim(i_{edit}, i)} \quad (5)$$

The output of *Icon2Code* will be a list of Android APIs that are ranked based on the scores returned by Formula (5). For instance, the API lists recommended for the examples given in Listing 2 are displayed in Listing 4. *Icon2Code* will only return top-$N$ APIs, where $N$ is another user-configurable parameter. As well as the top-N APIs recommended for the active callback method associated with an icon, *Icon2Code* will also provide API usage samples that are gathered from the actual implementations of the selected similar icons.

Listing 4: The recommended API list for the examples given in Listing 2, where N = 5.

```
// Example 1
<View: int getId()>
<MediaPlayer: boolean isPlaying()>
<MediaPlayer: void pause()>
<MediaPlayer: void setLooping(...)>
<MediaPlayer: void start()>

// Example 2
<Dialog: void <init>()>
<Dialog: void setCancelable(...)>
<Dialog: void setContentView(...)>
<Dialog: Window getWindow()>
<Window: boolean requestFeature(...)>
```

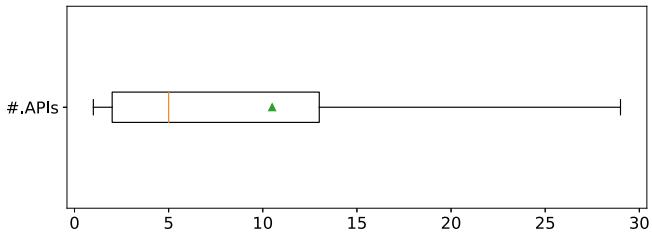



**Fig. 4.** The distribution of the number of APIs accessed by icon-bound GUI components.

**Implementation.** Our prototype tool *Icon2Code* is implemented in Java and on top of several well-known existing tools. JADX[1] is leveraged to disassemble Android APKs and convert the APK bytecode into Java source code. Gator [31], specifically its GUIHierarchyPrinterClient module, is used to infer callback methods for pre-identified icon-bound GUI components. *Icon2Code* leverages Soot [32] to construct call graphs for all the event handler callback methods and extract APIs invoked by these callback methods, as well as code snippets to be used as examples showing how APIs are accessed in practice.

## 4. Evaluation

We evaluate the effectiveness of *Icon2Code* by answering the following three research questions:

- **RQ1:** How accurate is *Icon2Code* in recommending API calling code for GUI components of Android apps under development?
- **RQ2**: Do the number of the most similar icons and their corresponding code implementations selected for learning impact *Icon2Code*'s performance?
- **RQ3**: To what extent do different weights of text/image similarities impact the performance of *Icon2Code*?
- **RQ4:** Will the performance of *Icon2Code* be impacted by the number of APIs accessed by icons selected for training?

### 4.1. Dataset

We need a quality dataset containing icons mapped to a set of APIs that are accessed after user interaction with the icons to support our experiments. Unfortunately, no such dataset has yet been released or made available and we had to build one from scratch. We leveraged the first DCM module of *Icon2Code* and applied it to a set of randomly selected Google Play apps, collected from AndroZoo [33]. DCM scans a given Android app to check if image files are provided. If so, it leverages code analysis to construct mappings from icons to their associated callback methods. If there are icon–callback pairs identified, we extract APIs accessed by those callback methods and record the results into the database if at least one API is collected.

We ran this process and built a benchmark database with 47,827 icons from approximately 5000 apps. Each of the icons in this database accesses at least one API. Fig. 4 presents the distribution of the number of API calls per icon. The median and average numbers are 5 and

[1] https://github.com/skylot/jadx.

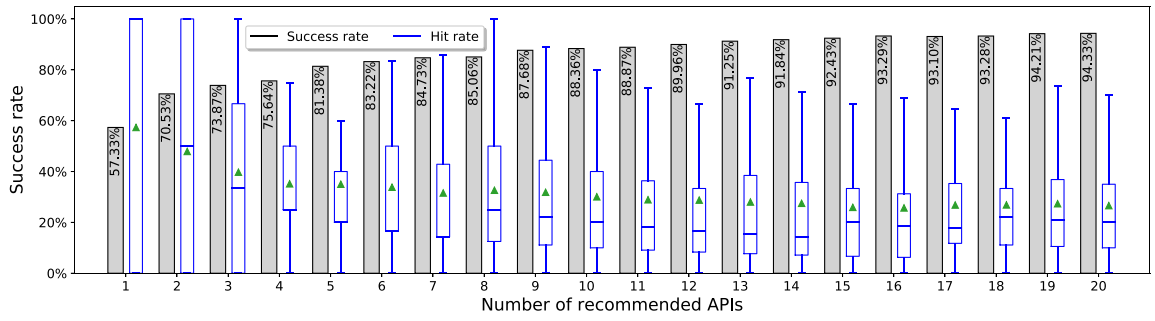



**Fig. 5.** Experimental results of *Icon2Code* in recommending API usages to icon-bound GUI components.

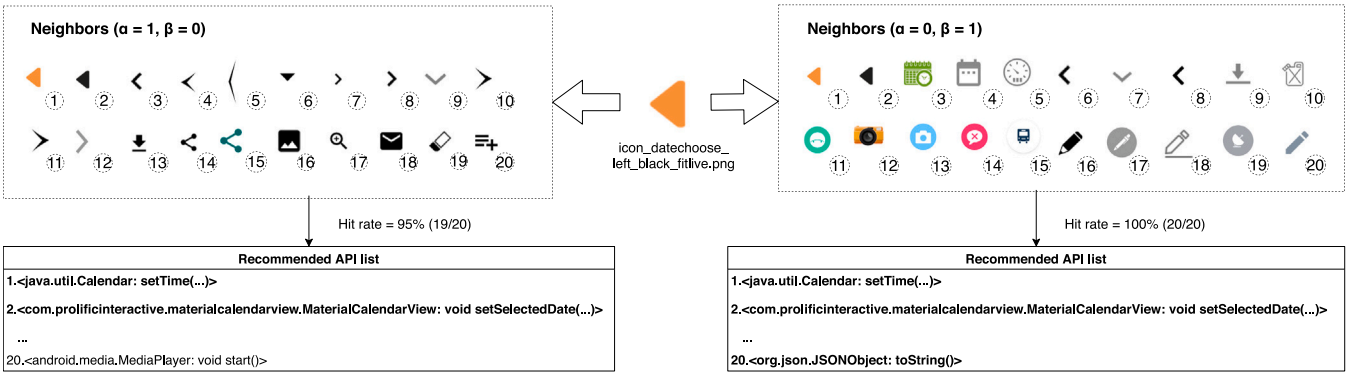



**Fig. 6.** A concrete example of recommending APIs for a target icon extracted from app *com.zjw.wearheart*.

10.5, respectively. This suggests that implementing each icon-related callback method is likely to be complicated, involving more than five API calls in over half of the cases. In an extreme case, the number of called APIs is as high as 30.

### 4.2. Evaluation metrics

Given an icon or GUI component and its callback method under development, the objective of *Icon2Code* is to recommend a ranked list of APIs (e.g., $N$ APIs) to help developers complete the implementation of the callback function. To assist in evaluating whether *Icon2Code* satisfies this purpose, we leverage the commonly used *success rate* and *hit rate* metrics to assess the usefulness and effectiveness of our approach. These two metrics, either applied to evaluate result@1 or result@N, have been recurrently leveraged by our fellow researchers to assess other code recommendation approaches [30,34].

The success rate metric has been frequently leveraged to evaluate the effectiveness of similar recommendation systems. For example, Nguyen et al. [30] leveraged it to evaluate performance of their method-to-API usage recommendation system. Given a set of icons $ICON$ under testing, for the callback functions under development of each icon $icon$, *Icon2Code* generates $N$ recommended APIs, i.e., $R_N(icon)$, to fulfill them. We consider that a recommendation is successful for icon $icon$ as long as at least one out of the $N$ APIs are in the Ground-Truth set $GT(icon)$. The success rate for $ICON$ can then be calculated via Formula (6), where $GT(icon)$ stands for the set of APIs actually accessed by the callback functions of $icon$, and $match_N(icon)$ is defined as the intersection of the recommended $N$ APIs and $GT(icon)$, i.e., $match_N(icon) = R_N(icon) \cap GT(icon)$.

$$success\ rate@N = \frac{count_{icon \in ICON}(|match_N(icon)| > 0)}{|ICON|} \times 100\% \quad (6)$$

The hit rate is another metric we leverage in this work to supplement the success rate metric to describe the ratio of the top $N$ recommended APIs matching $GT(icon)$:

$$hit\ rate@N = \frac{|match_N(icon)|}{N} \times 100\% \quad (7)$$

### 4.3. RQ1: Performance of Icon2Code

We aim to validate the performance and effectiveness of *Icon2Code*. Based on default parameters of *Icon2Code*, i.e., twenty neighbors ($m = 20$) and image only similarity calculation ($\alpha = 1, \beta = 0$), we perform experiments with a standard 10-fold cross-validation procedure. We use the dataset based on collected 47,827 icons and their corresponding set of APIs accessed by associated callback methods. We randomly divide our dataset into ten sets of 4,782 icons in each set. Nine sets are used as training set and the remaining one for testing. This process is then repeated ten times to confirm that each of the ten sets has been treated as a test set once. We finally apply the overall results of these ten validations to characterize the performance of *Icon2Code* in each experimental setting. In order to avoid the influence of icons from the same app on the experimental results, for all experiments in this work we narrow the selection range of neighbor candidates for each test icon to those gathered from different apps.

Fig. 5 shows our experimental results, including success rate as well as hit rate, concerning a different number of recommended APIs (i.e., success rate@N and hit rate@N, where $N \in [1, 20]$). Expectedly, the more number of APIs considered for recommendation, the higher the success rate of *Icon2Code* will be. When only one API is taken into account (i.e., success rate@1), *Icon2Code* can already achieve over 50% of the success rate. If we increase the number to 20, the success rate can reach over 94%, showing high performance to be applicable in practice.

Regarding the hit rate, as the number of considered APIs increasing, it first slightly declines and then tends to become stable. This suggests that (1) the top-recommended APIs have a high possibility to be the ones needed by the developers, and (2) more APIs will hit the ground truth if more APIs considered. This experimental result shows that our approach is useful in recommending APIs for assisting developers in implementing the callbacks of icon-bound GUI components.

**Case study.** We provide a concrete case study to demonstrate the effectiveness of *Icon2Code*. Fig. 6 presents a typical case, where the icon under development is used to rewind to the past during date selection

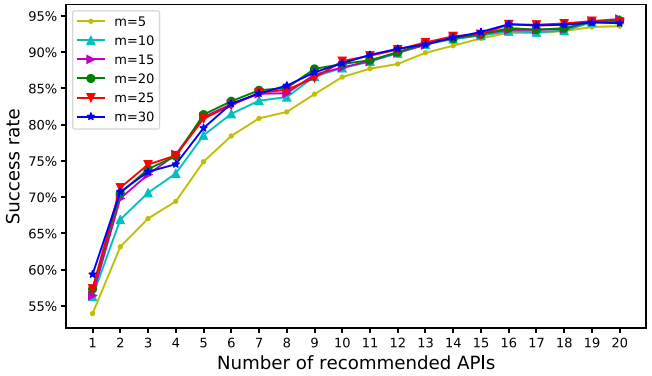



**Fig. 7.** The success rates obtained by altering the number of similar methods (i.e., parameter $m$).

in a calendar component. The top left box shows the 20 neighbors (based on image similarity) returned from the training database while the top right box shows the 20 neighbors obtained through text similarity. Due to our current simplistic implementation of image similarity calculation, not all neighbor icons are closely similar to the target icon. Nevertheless, the top-ranked icons such as (1)(2)(3) are very similar to the targeted one, and all of them are also relevant to date manipulation. Subsequently, these similar icons will dominate the selection process and allow *Icon2Code* to achieve a hit rate at 95%, i.e., by learning from the implementations of these 20 neighbors, *Icon2Code* is able to recommend 19 (out of 20) APIs that hit the ground truth list.

Notice that the un-hit API is related to the implementation of media players. This API is recommended because several icons in the neighbor list (i.e., (4)(10)(11)) are related to managing the media player. This result indicates that the neighbor list's quality is essential to the effectiveness of *Icon2Code* in recommending API usage code. This was a key motivation for adding a text-similarity strategy into *Icon2Code* (as described in Section 3.2. A better image similarity calculation strategy could also improve the performance of *Icon2Code*.

### 4.4. RQ2: Impact of the selected number of similar icons

In our second research question, we explore the impact of altering the number of similar icons (i.e., the parameter $m$) on the performance of *Icon2Code*. To this end, we design multiple sets of experiments considering different numbers of neighbors and perform them with default settings for other parameters e.g. when only image similarity is considered.

Fig. 7 shows experimental results with respect to different parameters, i.e., $m \in \{5, 10, 15, 20, 25, 30\}$ ($m = 5$ means that *Icon2Code* will only build the encoding matrix with 5 similar icons). Similar to our previous finding, as the number of APIs considered for recommendation $N$ increases, no matter which $m$ is considered, the success rate also increases. By comparing increasing trends, $m = 5$ achieves the worst performance, followed by $m = 10$, which achieves slightly higher performance but still clearly less than all the other settings. Interestingly, when increasing the value of $m$ to 15, the performance starts to converge, i.e., the performance does not significantly change any more while increasing the number of similar images $m$. The hit rate follows a similar pattern, as shown in Fig. 8. By increasing the value of $m$, the hit rates slightly increase as well and start to stabilize when $m$ reaches 15 or 20.

These results show that the number of selected icons indeed impacts the performance of *Icon2Code* when it is small. When the number reaches a certain threshold, the impact tends to be marginal. Furthermore, they also show that the default value ($m = 20$) is a suitable number for *Icon2Code* to recommend API usages for icon-bound GUI components.

### 4.5. RQ3: Impact of similarity calculation methods

We now explore the impact of the similarity calculation methods on the performance of *Icon2Code*, i.e., the value of $\alpha$ and $\beta$ discussed in Section. 3.2. By default, the weights for image similarity and text similarity $(\alpha, \beta)$ are set to be (1,0).This means that only image files are required for the calculation of similarity i.e., the text is optional. To evaluate the advantages of including text similarities, we now compare this default setting with another four settings formed by altering the weights, i.e., (0.8, 0.2), (0.5, 0.5), (0.2, 0.8), and (0, 1). Weights (0, 1) stand for the cases where only text similarity is considered for locating similar icons in the training set. All the other parameters of *Icon2Code* are kept the same to ensure a fair comparison.

Fig. 10 illustrates these experimental results. Surprisingly, the text-only setting achieves the best performance, and yet all the alternative experimental settings (involving text similarities) outperform (or achieve comparable results compared to) the default setting when only image similarities are considered. Similarly, concerning the hit rate, as demonstrated in Fig. 9, experimental settings involving text similarities only achieves a better hit rates than that of image similarities alone. The performance differences, nonetheless, for both success rate and hit rate are quite marginal.

This shows that developers can also resort to alternate text to help find similar event handler callback methods. In the absence of icon images during design, developers could potentially leverage our approach to select suitable icons. This result suggests that the similarity calculation module is critical to the success of *Icon2Code*. With a better image (icon) similarity calculation method, *Icon2Code* could likely achieve better performance. Since this is not our main contribution to this work, we leave it for future work.

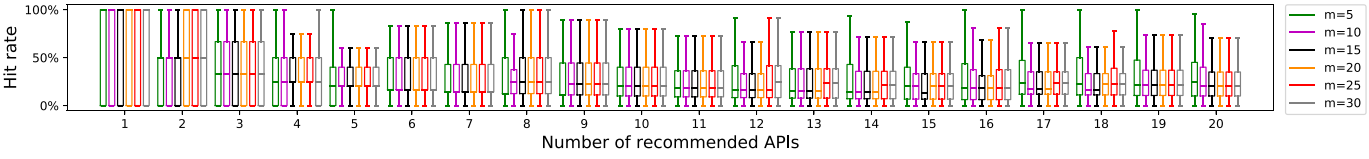



**Fig. 8.** The hit rates obtained by altering the number of similar methods.

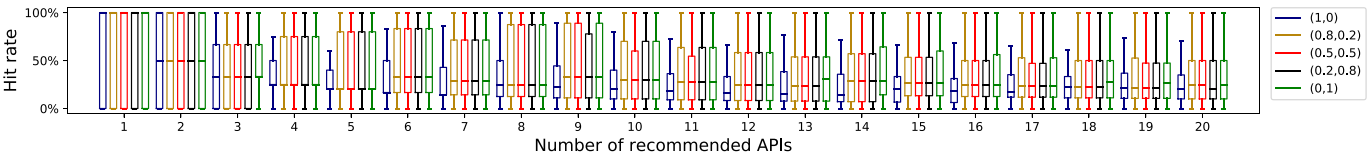



**Fig. 9.** The hit rates obtained by adjusting the similarity calculation strategies.

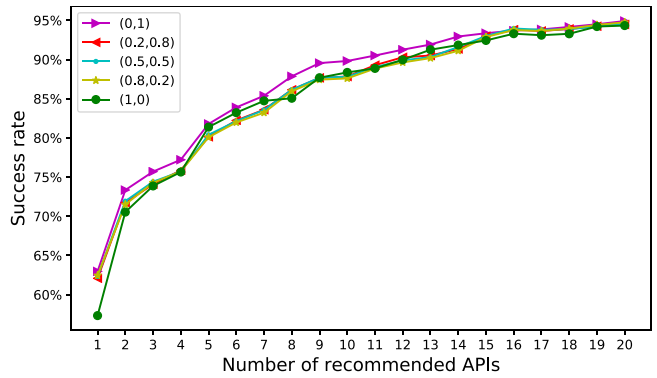



**Fig. 10.** The success rates obtained by adjusting the similarity calculation strategies (i.e., parameters $\alpha$ and $\beta$).

**Case Study Revisited.** We revisit the concrete example illustrated in Fig. 6. Interestingly, the 20 icon neighbors returned by text similarity do share some common icons with that obtained via image similarity. They also present several differences. The experiment results achieved via text similarity are as good as, or even slightly better than, the results achieved via image similarity. This shows that alternative text also provides useful information to locate similar icons so as to learn event handler code implementations for their associated GUI components. Future work should focus not only on improving the image similarity calculation methods but also on finding a smart way to combine the capabilities brought by both image and text similarities.

### 4.6. RQ4: Performance on different groups of training icons

Our last research question concerns the performance of the *Icon2Code* tool over different groups of training icons. In this work, we split the original training dataset into three groups: (1) All the icons that have no more than five APIs accessed by their associated callback methods, (2) All the icons that have over five but no more than 10 APIs accessed by their associated callback methods, and (3) All the icons that have over 10 APIs accessed by their associated callback methods. Following the same experimental setting, as discussed in Section 4.3, we re-launch *Icon2Code* on the aforementioned three training groups, respectively. Again, ten-fold cross-validation is leveraged in all three experiments.

Fig. 11 presents the experimental results. Interestingly, when comparing the results across the three experiments, the success rate increases from the first to the second groups and from the second to the third groups. For example, the success rate@5 of the three experiments are 74.56%, 81.6%, and 82.29%, respectively. This evidence suggests that increasing the number of APIs accessed by icons selected for training could improve the performance of *Icon2Code*. Furthermore, when looking at each of the experiments alone, *Icon2Code* achieves over 50% of success rate when only the first recommended item is concerned for all the three experiments. When the number of recommended APIs increases, the success rate also increases. Considering the hit rate, as expected, regardless of the groups, it will first slightly decrease when the number of API increases and then stabilizes. These experimental results are all in line with our previous experimental findings, confirming the effectiveness of *Icon2Code* in recommending API implementations for GUI components.

## 5. Discussion

We now perform sensitivity analysis for the threshold (i.e., 0.85) set in image similarity and the three weights used in calculating text similarities to examine if they are suitable for our work.

**Table 3**
Experimental results obtained by varying the threshold in image similarity.

| Criteria | 0.8 | 0.85 | 0.9 | 0.95 |
|---|---|---|---|---|
| Top-1 | 60.46% | 60.07% | 60.15% | 59.88% |
| Top-5 | 82.23% | 81.21% | 80.27% | 80.56% |
| Top-10 | 89.57% | 89.37% | 89.57% | 90.01% |
| Top-15 | 91.48% | 92.14% | 92.77% | 92.8% |
| Top-20 | 93.58% | 94.04% | 94.46% | 94.73% |

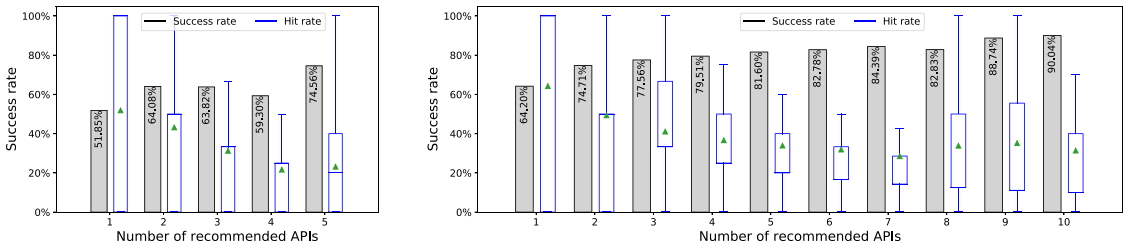
**Threshold in image similarity.** In order to perform sensitivity analysis on the threshold in image similarity calculation, based on the parameters of twenty neighbors ($m = 20$) and image only similarity calculation ($\alpha = 1, \beta = 0$), we perform experiments on the same dataset as Section 4, where standard 10-fold cross-validation procedure is taken into account. We vary the threshold from 0.85 to 0.95 with an interval at 0.05. Table 3 summarizes the experimental results. It can be observed from the results that increasing the similarity threshold may not necessarily yield better results. Indeed, when threshold 0.85 is considered, the performance at Top-1 and Top-5 is even slightly lower than the results achieved by setting the threshold at 0.85. Nevertheless, overall, the adjustments of the similarity threshold have little impact on the experimental results. This evidence indicates that our approach is not sensitive to the image similarity threshold. We hypothesis that this insensitivity could be related to the fact that we have only leveraged traditional image similarity calculation algorithms, which may not be capable of characterizing the images' semantics. As for our future work towards verifying this hypothesis, we will resort to deep learning models to measure images' similarities.

**The three weights in text similarity.** Similar to the sensitivity analysis for threshold in image similarity calculation, we further conduct experiments to evaluate the sensitivities of the weights set to calculate text similarities. In this setting, the parameters are adjusted to twenty neighbors ($m = 20$) and text only similarity calculation ($\alpha = 0, \beta = 1$). We perform 7 sets of experiments, using different values of $(w_1, w_2, w_3)$ in Eq. (3), and the results are shown in Table 4. The experimental results show that there is no clear winner among the three alternative texts. Therefore, we set the three alternative texts the same weight to make *Icon2Code* more versatile, aiming to avoid the impact of corner cases such as one type of alternative text in some apps not provided by app developers.

### 5.1. Threats to validity

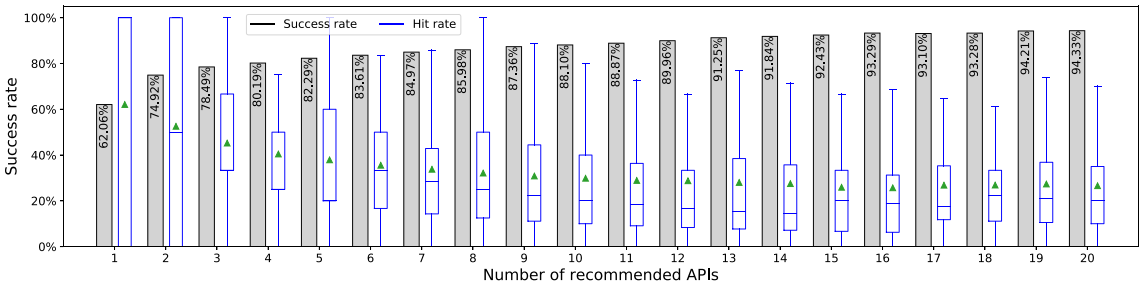
**Threats to construct validity.** *Icon2Code* is implemented based on several static analysis frameworks, including Soot [35] and Gator [31,36,37]. Their reliability defects hence could propagate to *Icon2Code*, thus introduce threats to the effectiveness of our approach. Nonetheless, the above frameworks have been demonstrated to be useful by various state-of-the-art works [38–41] and hence the potential threats should be limited.

**Threats to internal validity.** The main internal threat is that we apply simulated experimental settings for evaluation rather than studying real-world recommendation scenarios. Following the state-of-the-art [30], we mitigate this threat by forming a large-scale dataset and employing 10-fold cross-validation to reduce the impact of contingency. We used commonly agreed heuristic evaluation metrics for our experiments with *Icon2Code*. These may not indicate the actual performance of *Icon2Code* in practice, and hence user studies are needed to evaluate its effectiveness in real-world code recommendation. *Icon2Code* cannot yet handle the situation where several icons share the same callback methods. Furthermore, in some rare cases, the harvested images do not represent the purpose of the GUI components e.g., icons are provided as background. This may introduce unrelated cases to our training database so as to impact the final recommendation results. We plan to invent a means to filter out such irrelevant icons and thereby to improve the performance of our approach.




(a) Performance on samples with 1-5 ground-truth APIs

(b) Performance on samples with 6-10 ground-truth APIs




(c) Performance on samples with 11-20 ground-truth APIs

**Fig. 11.** Experimental results of *Icon2Code* in recommending API usages to icon-bound GUI components with different numbers of accessed APIs.

**Table 4**
Experimental results obtained by varying the three weights in text similarity.

| Criteria | (0.33,0.33,0.33) | (1,0,0) | (0,1,0) | (0,0,1) | (0.5,0.5,0) | (0.5,0,0.5) | (0,0.5,0.5) |
|---|---|---|---|---|---|---|---|
| Top-1 | 62.89% | 61.49% | 64.9% | 58.59% | 62.93% | 61.55% | 65.1% |
| Top-5 | 81.77% | 79.88% | 83.17% | 75.05% | 81.83% | 79.82% | 83.2% |
| Top-10 | 89.95% | 88.45% | 88.54% | 88.9% | 90% | 88.23% | 88.69% |
| Top-15 | 93.49% | 93.24% | 93.42% | 93.93% | 93.62% | 93.19% | 93.53% |
| Top-20 | 94.85% | 95.24% | 95.02% | 96% | 95.04% | 95.14% | 94.97% |

**Threats to external validity.** A major external threat lies in the random selection of mobile apps for establishing our dataset. These may not generally represent all the available apps in the Android ecosystem. We strived to mitigate this threat by randomly selecting real-world apps released in the official Google Play store, and conducting further analysis and screening to retain those that meet our needs. Additionally, at present, app obfuscation may be applied to some apps in our dataset which may confuse icon to event handler code mappings. We did not consider it in this work. Nevertheless, as we are primarily interested in learning API usage, which would not be affected by simple obfuscation strategies such as method renaming) [42].

### 5.2. Limitations and future work

It is known that recommendation systems often suffer from the so-called *cold start* problem. This concerns the issue of the system's inability to draw any inferences for users or items about which it has not yet collected sufficient information [43]. Potentially, our approach could also suffer from such a threat. To better understand to what extent our approach is impacted by the *cold start* problem, we conduct an empirical investigation by varying the number of icons our approach could learn from the training dataset. We form our training set with the following number of available icons: {50, 100, 500, 1000, 1500}. We then re-run the experiments discussed in Section 4.3 with all the other parameters kept to their default values. Table 5 summarizes our experimental results. As expected, the success rate increases when enlarging the size of the training dataset. When setting the number of icons included for training at 1500, our recommendation approach's success rate can already exceed 80% at Top-5, 90% at Top-10, and even 95% if Top-20 is considered. This experimental result shows that the *cold start* problem indeed impacts the performance of our approach. However, such an impact could be significantly mitigated if more training apps are prepared. To the best of our knowledge, such a requirement is not difficult to achieve as it is relatively easy to collect more Android apps. For example, the well-known AndroZoo dataset [33] has collected over 10 million real-world Android apps ready for adoption by our approach. In our future work, we plan to enlarge our training set to provide more accurate API recommendations for developers to implement the logic behind each icon.

**Table 5**
Experimental results obtained by varying the size of the training dataset.

| Criteria | 50 | 100 | 500 | 1000 | 1500 |
|---|---|---|---|---|---|
| Top-1 | 64.44% | 64.69% | 66.91% | 68.68% | 69.5% |
| Top-5 | 83.02% | 83.16% | 83.19% | 83.6% | 83.62% |
| Top-10 | 90.41% | 91.46% | 91.52% | 91.73% | 92.81% |
| Top-15 | 94.75% | 94.8% | 95.48% | 95.75% | 96.06% |
| Top-20 | 95.24% | 95.54% | 95.23% | 96.76% | 96.9% |

We have not distinguished between Android APIs and third-party library APIs when recommending API usages for icon-bound event handler callback methods. Our approach could be leveraged to recommend third-party libraries for implementing icons' callback methods. Third-party library APIs may provide enough information to infer the actual libraries, and even the distinct versions of the libraries, to some extent. Library recommendation has been a hot topic in the software engineering community [34]. For example, He et al. [44] have proposed an approach to predict diversified third-party libraries for helping developers implement mobile apps. This approach could be leveraged to supplement ours so as to predict the right third-party libraries and code snippets to use to help in implementing mobile GUI components.

Apart from recommending API usage to icon-bound GUI components, our approach could also be leveraged to recommend icons to GUI designers. Given an alternative text describing the purpose of the icon (not yet designed), the returned icon neighbors (samples) in the second module of *Icon2Code* could be leveraged by designers to create suitable icons.

Furthermore, we plan to work on an improved similarity calculation module for *Icon2Code*, aiming to invent more effective approaches to locate similar icons. At the moment, we have leveraged Levenshtein distance to calculate the text-similarity in this paper, which is a commonly-used string metric for measuring the difference between two sequences. As for future work, we consider introducing improved algorithms such as deep learning networks to better calculate such similarities. Additionally, for calculating the similarities of icon images, we plan to also leverage state-of-the-art computer vision techniques to better find the most suitable neighbors of the input icon so as to achieve more relevant code implementation learning.

Some of the event handler callback methods, although rare, could be bound to several GUI components. Developers often use conditional judgments to define which specific code segments are related to which icon. At the moment, this type of setting is agnostic by our approach and which could hence lead to inaccurate results. It is also possible that the categorizes of UI widgets may have some impacts on the performance of our approach, e.g., our approach only works well on certain types of buttons, or image views, etc. As for our future work, we plan to take these cases into consideration when improving our approach.

We intend to integrate our prototype implementation as a plugin into Android Studio, the default IDE recommended for app developers. We then want to recommend icon event handler API usage during the development phase of given Android apps that require much coding work for icon-bound GUI components. We want to carry out user studies of *Icon2Code* in this context to determine if its event handler code recommendations prove useful for large scale, complex Android development projects.

Finally, to better evaluate the effectiveness of our approach, we plan to conduct a large-scale user study involving carefully designed tasks and representative user skills [30]. We also plan to improve the usability of our approach to alleviate unnecessary noises introduced to the user study approach.

## 6. Related work

We summarize critical related work from three aspects, i.e., GUI component analysis of mobile apps, recommendations in Android development, and collaborative filtering approaches applied in software engineering.

### 6.1. GUI analysis in Android

To help developers better implement GUI components, several tools have been developed [19,45,46] to assist the transition from UI design images to GUI implementations. For example, Chen et al. [19] propose a deep learning-based technique trained with the UI design and GUI implementation knowledge learned from existing apps to convert UI requirements into a hierarchy of GUI components. Unlike these works, which focus on the code implementation of the UI design images, *Icon2Code* aims to recommend API usages for Android GUI components' event handlers i.e., their associated callback methods.

Rountev et al. [31] target static object reference analysis to model GUI-associated Android objects, their flow through the application, and their interactions with each other via the abstractions defined by the Android platform. In our work, we go one step further to focus on the specific code implementation of callbacks related to the GUI components. We also leverage traditional recommendation algorithm to provide developers with references and suggestions to help them achieve rapid development.

Xiao et al. [41] present a framework that leverages program analysis techniques to associate icons and GUI widgets and classifies the associated icons into eight sensitive categories for Android apps. Xi et al. [47] propose DeepIntent, a framework that associates the icons and contextual texts with GUI widgets' program behaviors. It infers the GUI widgets' permission uses based on the program behaviors, synthetically consolidating program analysis, and deep learning techniques to identify intention-behavior disparities. Although the above two works are related to the analysis of GUI components and program behaviors, our work is different. Their work represents the program behaviors through the permissions used by the mobile apps, while our work focuses more on code level API invocations called by the callbacks methods.

### 6.2. Recommendation in Android development

Researchers have devoted much effort in facilitating Android API recommendations to support mobile app development. This is because Android apps development relies extensively on API frameworks and libraries. Some works attempt to provide relevant suggestions on using third-party libraries [44,48,49]. Others center on delivering real-time recommendations during the development, such as giving parameter values as suggestions in similar programming scenarios [50], or providing Android APIs and their usage patterns for assisting in developers' work [11,13,51].

Gu et al. [52] generate API usage sequences based on natural language query through a deep learning-based approach for the purpose of code search. Likewise, Jiang et al. [53] propose an approach leveraging multi-aspect features to generate code snippets as recommendations, such as text, topic, and the number of lines, etc. There is a module in *Icon2Code* that utilizes a simple short text to locate similar neighbors to facilitate subsequent recommendations. We believe the aforementioned approaches could supplement this module to enable *Icon2Code* to achieve a higher performance.

Yuan et al. [18] initially concentrate on the demand of recommending event callbacks in Android application development and submit an approach to support both functional APIs and the event callbacks that need to be overridden. They extended this work by establishing a large Android-specific API database indicating the associations among diverse functionalities and APIs [12]. What is different from their work to ours, aside from the GUI component mapping, is that callbacks in our work are the objects that need to be fulfilled, that is, are playing the role of **users** in the recommendation system, rather than **items** in theirs.

### 6.3. Collaborative filtering in software engineering

Collaborative filtering techniques are broadly employed in software engineering to support many different recommendation systems. Thung et al. [54] integrate association rule mining and user-based collaborative filtering and introduce a technique to recommend likely related libraries to aid developers in exploit third-party libraries. Similarly, Yu et al. [34] propose an approach that blends Latent Dirichlet Allocation (LDA) and collaborative filtering to give suggestions about third-party libraries for mobile apps. He et al. [44] propose an approach that leverages Matrix Factorization, a classic collaborative filtering based prediction approach, for recommending useful third-party libraries to developers. These are for general app code usage, rather than for GUI event handler code implementation as in our work.

In terms of applications at the code level, Nguyen et al. [30] present a context-aware collaborative filtering based algorithm to recommend Java method invocations. We focus on providing much more targeted API and usage pattern recommendations for callbacks related to specific GUI components.

## 7. Summary

We have proposed a prototype tool *Icon2Code* to recommend API calling code to assist Android app developers in implementing the callback functions of iconic GUI components. *Icon2Code* leverages icon image files and their alternative text to locate similar icons that are closest to the active icon under development. It then employs a collaborative filtering algorithm with encoding matrix and rating algorithms to obtain an output of recommended APIs to call in the event handler code, as well as usage samples from existing apps. Our experimental results using a new dataset of almost 50,000 icon event handler implementations have demonstrated that *Icon2Code* is effective in recommending such event handler code API usage for Android developers.

## CRediT authorship contribution statement

**Yanjie Zhao:** Conceptualization, Methodology, Software, Data curation. **Li Li:** Supervision, Conceptualization, Methodology, Writing - review & editing. **Xiaoyu Sun:** Software, Data curation. **Pei Liu:** Software, Data curation. **John Grundy:** Supervision, Writing - review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments that have led to substantial improvements in this manuscript. This work was partly supported by the Australian Research Council (ARC) under a Laureate Fellowship project FL190100035, a Discovery Early Career Researcher Award (DECRA) project DE200100016, and a Discovery project DP200100020.

## References

[1] Amanda Short, Standing Out From The Crowd: Improving Your Mobile App With Competitive Analysis, Smashing Magazine, 2017, URL: https://www.smashingmagazine.com/2017/12/improving-mobile-app-competitive-analysis/. [Online; accessed 20-July-2020].

[2] H. Cai, Z. Zhang, L. Li, X. Fu, A large-scale study of application incompatibilities in android, in: The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019), 2019.

[3] L. Li, T.F. Bissyandé, H. Wang, J. Klein, CiD: Automating the detection of API-related compatibility issues in android apps, in: The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018), 2018.

[4] J. Gao, L. Li, P. Kong, T.F. Bissyandé, J. Klein, Understanding the evolution of android app vulnerabilities, IEEE Trans. Reliab. (TRel) (2019).

[5] J. Gao, P. Kong, L. Li, T.F. Bissyandé, J. Klein, Negative results on mining crypto-API usage rules in android apps, in: The 16th International Conference on Mining Software Repositories (MSR 2019), 2019.

[6] P. Liu, L. Li, Y. Yan, M. Fazzini, J. Grundy, Identifying and characterizing silently-evolved methods in the android API, in: The 43rd ACM/IEEE International Conference on Software Engineering, SEIP Track (ICSE-SEIP 2021), 2021.

[7] L. Li, J. Gao, T.F. Bissyandé, L. Ma, X. Xia, J. Klein, CDA: Characterising deprecated android APIs, Empir. Softw. Eng. (EMSE) (2020).

[8] L. Li, T. Riom, T.F. Bissyandé, H. Wang, J. Klein, Y. Le Traon, Revisiting the impact of common libraries for android-related investigations, J. Syst. Softw. (JSS) (2019).

[9] L. Li, T.F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, Y. Le Traon, Static analysis of android apps: A systematic literature review, Inf. Softw. Technol. (2017).

[10] P. Kong, L. Li, J. Gao, K. Liu, T.F. Bissyandé, J. Klein, Automated testing of android apps: A systematic literature review, IEEE Trans. Reliab. (2018).

[11] T.T. Nguyen, H.V. Pham, P.M. Vu, T.T. Nguyen, Recommending API usages for mobile apps with hidden markov model, in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 795–800.

[12] W. Yuan, H.H. Nguyen, L. Jiang, Y. Chen, J. Zhao, H. Yu, API recommendation for event-driven Android application development, Inf. Softw. Technol. 107 (2019) 30–47.

[13] H. Niu, I. Keivanloo, Y. Zou, API usage pattern recommendation for software development, J. Syst. Softw. 129 (2017) 127–139.

[14] M.M. Rahman, C.K. Roy, D. Lo, Rack: Automatic api recommendation using crowdsourced knowledge, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, IEEE, 2016, pp. 349–359.

[15] H. Zhong, T. Xie, L. Zhang, J. Pei, H. Mei, MAPO: Mining and recommending API usage patterns, in: European Conference on Object-Oriented Programming, Springer, 2009, pp. 318–343.

[16] W. Yang, M.R. Prasad, T. Xie, A grey-box approach for automated GUI-model generation of mobile applications, in: International Conference on Fundamental Approaches to Software Engineering, Springer, 2013, pp. 250–265.

[17] S. Gao, L. Liu, Y. Liu, H. Liu, Y. Wang, API recommendation for the development of Android App features based on the knowledge mined from App stores, Sci. Comput. Program. 202 (2021) 102556.

[18] W. Yuan, H.H. Nguyen, L. Jiang, Y. Chen, LibraryGuru: API recommendation for Android developers, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2018, pp. 364–365.

[19] C. Chen, T. Su, G. Meng, Z. Xing, Y. Liu, From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 665–676.

[20] D.D. Perez, W. Le, Generating predicate callback summaries for the android framework, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2017, pp. 68–78.

[21] W. Song, X. Qian, J. Huang, EHBDroid: Beyond GUI testing for Android applications, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 27–37.

[22] V. Avdiienko, K. Kuznetsov, I. Rommelfanger, A. Rau, A. Gorla, A. Zeller, Detecting behavior anomalies in graphical user interfaces, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 201–203.

[23] E. Rublee, V. Rabaud, K. Konolige, G. Bradski, ORB: An efficient alternative to SIFT or SURF, in: 2011 International Conference on Computer Vision, Ieee, 2011, pp. 2564–2571.

[24] M. Datar, N. Immorlica, P. Indyk, V.S. Mirrokni, Locality-sensitive hashing scheme based on p-stable distributions, in: Proceedings of the Twentieth Annual Symposium on Computational Geometry, 2004, pp. 253–262.

[25] D.R. Kaeli, P. Mistry, D. Schaa, D.P. Zhang, Heterogeneous Computing with OpenCL 2.0, Morgan Kaufmann, 2015.

[26] E.S. Ristad, P.N. Yianilos, Learning string-edit distance, IEEE Trans. Pattern Anal. Mach. Intell. 20 (5) (1998) 522–532.

[27] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, in: Soviet Physics Doklady, Vol. 10, 1966, pp. 707–710.

[28] S. Sarkar, D. Das, P. Pakray, A. Gelbukh, JUNITMZ at SemEval-2016 task 1: Identifying semantic similarity using Levenshtein ratio, in: Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016), 2016, pp. 702–705.

[29] J.B. Schafer, D. Frankowski, J. Herlocker, S. Sen, Collaborative filtering recommender systems, in: The Adaptive Web, Springer, 2007, pp. 291–324.

[30] P.T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, M. Di Penta, Focus: A recommender system for mining api function calls and usage patterns, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 1050–1060.

[31] A. Rountev, D. Yan, Static reference analysis for GUI objects in Android software, in: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2014, pp. 143–153.

[32] S. Arzt, S. Rasthofer, E. Bodden, The soot-based toolchain for analyzing android apps, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2017, pp. 13–24.

[33] L. Li, J. Gao, M. Hurier, P. Kong, T.F. Bissyandé, A. Bartel, J. Klein, Y. Le Traon, Androzoo++: Collecting millions of android apps and their metadata for the research community, 2017, arXiv preprint arXiv:1709.05281.

[34] H. Yu, X. Xia, X. Zhao, W. Qiu, Combining collaborative filtering and topic modeling for more accurate android mobile app library recommendation, in: Proceedings of the 9th Asia-Pacific Symposium on Internetware, 2017, pp. 1–6.

[35] P. Lam, E. Bodden, O. Lhoták, L. Hendren, The soot framework for java program analysis: a retrospective, in: Cetus Users and Compiler Infastructure Workshop (CETUS 2011), Vol. 15, 2011, p. 35.

[36] S. Yang, D. Yan, H. Wu, Y. Wang, A. Rountev, Static control-flow analysis of user-driven callbacks in android applications, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 89–99.

[37] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, A. Rountev, Static window transition graphs for Android, Autom. Softw. Eng. 25 (4) (2018) 833–873.

[38] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, Acm Sigplan Not. 49 (6) (2014) 259–269.

[39] L. Li, A. Bartel, T.F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. McDaniel, Iccta: Detecting inter-component privacy leaks in android apps, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1, IEEE, 2015, pp. 280–291.

[40] R. Coppola, M. Morisio, M. Torchiano, Evolution and fragilities in scripted gui testing of android applications, in: Proceedings of the 3rd International Workshop on User Interface Test Automation, ACM, 2017.

[41] X. Xiao, X. Wang, Z. Cao, H. Wang, P. Gao, Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 257–268.

[42] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, K. Zhang, Understanding Android obfuscation techniques: A large-scale investigation in the wild, in: International Conference on Security and Privacy in Communication Systems, Springer, 2018, pp. 172–192.

[43] B. Lika, K. Kolomvatsos, S. Hadjiefthymiades, Facing the cold start problem in recommender systems, Expert Syst. Appl. 41 (4) (2014) 2065–2073.

[44] Q. He, B. Li, F. Chen, J. Grundy, X. Xia, Y. Yang, Diversified third-party library prediction for mobile app development, IEEE Trans. Softw. Eng. (2020).

[45] S.P. Reiss, Y. Miao, Q. Xin, Seeking the user interface, Autom. Softw. Eng. 25 (1) (2018) 157–193.

[46] T. Beltramelli, pix2code: Generating code from a graphical user interface screenshot, in: Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, 2018, pp. 1–6.

[47] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu, et al., DeepIntent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 2421–2436.

[48] Z. Ma, H. Wang, Y. Guo, X. Chen, LibRadar: fast and accurate detection of third-party libraries in Android apps, in: Proceedings of the 38th International Conference on Software Engineering Companion, 2016, pp. 653–656.

[49] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, W. Huo, LibD: scalable and precise third-party library detection in android markets, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 335–346.

[50] L. Li, T.F. Bissyandé, J. Klein, Y. Le Traon, Parameter values of Android APIs: A preliminary study on 100,000 apps, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, IEEE, 2016, pp. 584–588.

[51] T.T. Nguyen, H.V. Pham, P.M. Vu, T.T. Nguyen, Learning API usages from bytecode: a statistical approach, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 416–427.

[52] X. Gu, H. Zhang, D. Zhang, S. Kim, Deep API learning, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 631–642.

[53] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, X. Luo, Rosf: Leveraging information retrieval and supervised learning for recommending code snippets, IEEE Trans. Serv. Comput. 12 (1) (2016) 34–46.

[54] F. Thung, D. Lo, J. Lawall, Automated library recommendation, in: 2013 20th Working Conference on Reverse Engineering (WCRE), IEEE, 2013, pp. 182–191.