Improving the Efficiency of Mobile User Interface Development through Semantic and Data-Driven Analyses

Jieshan Chen

A thesis submitted for the degree of
Doctor of Philosophy of
The Australian National University

October 2022

Except where otherwise indicated, this thesis is my own original work.

Jieshan Chen
6 October 2022

To my parents, my big brother and my sisters.

Acknowledgments

First of all, I would like to thank my supervisors, Prof. Zhenchang Xing and Prof. Chunyang Chen. Thanks for everything they have done for me. Both of them are really patient and supportive. All the meetings with them were always inspiring and filled me with excitement and energy. The constructive advise they gave me helps me to properly deal with all the depressions and confusion I had during my research, and encourages me to steadily proceed with my research. Without their guidance, I could not have done so much.

I would also like to thank my colleague, Dehai Zhao, who helped me go through all the ups and downs during my Ph.D. journal. It is hard to count the number of times when I felt stressed and hopeless, and sought help from Dehai. He is a reliable and considerate friend, who can always understand my feeling and enlighten me. I feel extremely happy to have such a great friend.

I also like to express my appreciation for my friends and other colleagues in our lab. Thank Xiaoyin Chen for bringing me so much fun. Thank Xiaoxue Ren, Jiamou Sun and Mulong Xie for all the discussions and supports.

I enjoyed working with my mentor and my colleagues during my Apple internship, especially Xiaoyi Zhang, Amanda Swearngin, Jason Wu, Titus Barik and Jeffrey Nichols. I was in a bad time during that period, struggling to find a way out. It was Xiaoyi who gave me the opportunity to work in a different environment, and helped me go through the tough time. During this internship, I became a more independent and confident person. I learned to properly understand my own strengths and weaknesses.

Last but not least, the long-term support from my family means a lot. It is my big brother, Guibin, who guided me to pursue the Ph.D. degree, and gave me valuable suggestions at each important moment of my life. I should give him credit for all of my achievements. I also want to thank my sisters, Yanshan and Guishan, who gave me many companions and emotional supports. Many thanks to my parents, who help me build up a strong will and support me all the time.

Abstract

Having millions of mobile applications from Google Play and Apple's App store, the smartphone is becoming a necessity in our daily life. People could access a wide variety of services by using the mobile application, between which user interfaces (UIs) work as an important proxy. A well-designed UI makes an application easy, practical, and efficient to use. However, due to the rapid application iteration speed and the shortage of UI designers, developers are required to design the UIs and implement them in a short time. As a result, they may be unaware of or compromise some important factors related to usability and accessibility during the process of developing user interfaces of mobile applications. Therefore, efficient and useful tools are needed to enhance the efficiency of the development of user interfaces.

In this thesis, I proposed three techniques to improve the efficiency of designing and developing user interfaces through semantic and data-driven analyses. First, I proposed a UI design search engine to help designers or developers quickly create trendy and practical UI designs by exposing them to UI designs in real applications. I collected a large-scale UI design dataset by automatically exploring UIs from top-downloaded Android applications, and designed an image autoencoder-based UI design engine to enable finer-grained UI design search.

Second, during the process of understanding the real UIs implementation, I found that existing applications have a severe accessibility issue of lacking labels for image-based buttons. Such an issue will hinder the blind users to access the key functionalities on UIs. As blind users need to rely on screen readers to read content on UIs, it requires the developers to set up appropriate labels for image-based buttons. Therefore, I proposed LabelDroid, which aims to automatically generate labels (i.e., the content description) of image-based buttons to assist the implementation process of UIs.

Finally, as the above techniques all require the view hierarchical information, which contains the bounds and type of contained elements, to achieve the goals, some UIs may fail to have a complete view hierarchy or do not have such information. For example, UIs in the design-sharing platforms do not have any metadata about the elements. To do this, I conducted the first large-scale empirical study on evaluating existing object detection methods of detecting elements in UIs. By understanding the unique characteristics of UI elements and UIs, I proposed a hybrid method to boost the accuracy and precision of detecting elements on user interfaces. Such a fundamental method can be beneficial to many downstream applications, such as UI design search, UI code generation, and UI testing.

In conclusion, I proposed three techniques to enhance the efficiency of designing and developing the user interfaces on mobile applications through semantic and data-driven analyses. Such methods could easily generalize to a broader scope, such

as user interfaces of desktop apps and websites. I expect my proposed techniques and the understanding of user interfaces can facilitate the following research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smartphones are becoming a necessity in our daily life. Millions of mobile applications from Google Play store and Apple App store provide rich functionalities and services to end-users to facilitate our life, between which graphical user interfaces (GUIs) work as an important proxy. A well-designed GUI makes an application easy, practical and efficient to use, which significantly affects the success of the application and the loyalty of its users [1; 2; 3]. For example, in the competitive mobile application market, the design of an application's GUI, or even its icon, has become crucial for distinguishing an application from competitors, attracting user downloads, reducing users' complaints and retaining users [4; 5; 6], and thus make the company thrive. For end-users, user-friendly applications can enrich and facilitate their lives. However, the design and implementation of these user interfaces involve many challenges and inefficiency.

The development cycle of a mobile application involves many roles, including designers, developers, and end-users. Designers design every user interface (UI) in the application, choosing the appropriate widgets and icons, organizing them in a reasonable and easy-to-understand layout, and adding attractive and comfortable visual effects to every widget and the whole UI [7]. The developers then implement these designs using programming languages. They not only need to precisely implement the UI design from the designer, but also need to consider some "invisible" aspects that are not presented in the UI design. For example, developers need to choose a suitable layout element to group related UI elements on user interfaces while such layout elements are invisible in the pixel image from designers [8]. In addition, developers may also need to consider adding an accessibility label to the image-based buttons to ensure the accessibility of the minority (e.g. the blind users) [9; 10]. Moreover, due to the lack of sufficient design knowledge, developers need to iteratively communicate with the designers to understand requirements and explore all information that is implicitly expressed in the complicated design [11]. Apart from the above situation, developers often have to play the designer role in software development, especially in the start-up companies and open-source projects. In this thesis, we aim to identify the problems during the mobile user interface development process, and try to tackle with the identified problems through semantic and data-driven analyses.

1.1 Motivations and Goals

1.1.1 UI Design Search Engine

Designing a UI is not a trivial task. This process needs no only specific knowledge of design principles and guidelines (e.g., Android Material Design [12], iOS Human Interface Guidelines [13]), but also the understanding of design space which has the great variations in visual components that can be potentially used, their layout options, and visual effect choices. However, many developers often have to work as designers. For example, in a survey of more than 5,700 developers [14], 51% respondents reported that they do not have much UI design training but they work on UI design tasks, more so than other development tasks. While they can find some initial inspiration by browsing high-rated designs in design sharing platforms or use keywords to search the wanted design, these methods could only provide coarse-grained instructions that may not meet their needs.

As the advanced UI design search can consider UI designs as images, a naive solution could be searching UI designs by image similarity of certain image features such as color histogram [15] or Scale-Invariant Feature Transform (SIFT) [16]. Although such image features are useful for measuring image similarity, they are agonistic of the visual semantics (visual components and their compositions) of a GUI. As such, image-wise similar UI designs are very likely design-wise irrelevant. Alternatively, one can heuristically match individual visual components based on their type, position and size for measuring the similarity of two UI designs [17; 18; 19]. However, such methods are restricted by the pre-defined component matching rules, and is sensitive to the cut-off matching thresholds. Furthermore, individual component-matching heuristics often retrieve many irrelevant UI designs, because individual component matching cannot effectively encode the visual composition of components in a GUI as a whole. An effective mechanism is needed to support such developers to explore and learn about the UI design space in their UI design work.

1.1.2 Enhancing the Accessibility

Once the UIs are designed for a mobile application, developers then need to implement them using programming language. However, while the design provides cues on what elements are present and how should they position on the UIs, many important attributes related to accessibility are not visible on the UIs and would be easily compromised and ignored by the developers. For example, a well-designed UI may contain some icons, like a trash icon. While users can easily understand that by clicking that icon, a delete action will be performed. However, for people with disabilities, especially for the blind users, they need additional UI cues to understand that icons. According to the World Health Organization(WHO) [20], it is estimated that approximately 1.3 billion people live with some form of vision impairment globally, of whom 36 million are blind. Compared with the normal users, they may be more eager to use the mobile apps to enrich their lives, as they need those apps to represent their eyes. Ensuring full access to the wealth of information and services provided by mobile apps is a matter of social justice [21].

While developers can add the additional UI cues, i.e., the accessibility label, to the source code to fix the issues, many apps remain a serious accessibility issue. Based on our empirical study in Section 5.2, more than 77% apps out of 10,408 apps miss such labels for the image-based buttons, resulting in the blind's inaccessibility to the apps. Some existing work tries to help developers spot the potential accessibility issues [22; 23; 24; 25], but none of them can help fix the label-missing problem. An efficient and useful automated tool is needed to generate the labels for enhancing the accessibility.

1.1.3 Recognizing UI Elements.

Detecting Graphical User Interface (GUI) elements in GUI images is a domain-specific object detection task. It supports many software engineering tasks, such as GUI automation and testing [26; 27; 28; 29], supporting advanced GUI interactions [30; 31], GUI search [32; 17], and code generation [33; 34; 8]. By recognizing UI elements, the aforementioned tasks can be extended to UIs without metadata, i.e., the coordinates and types of UI elements on UIs. For example, we could include the UI design in design sharing platforms in the search engine, and update with the most trendy UI design constantly. In addition, while some metadata exist, it may not be always accurate. For example, Zhang et al. [35] also found that 59% of screens contain some UI elements that are not in the metadata, and 94% of their investigated apps have at least one such screen. Existing studies for GUI element detection directly borrow the mature methods from computer vision (CV) domain, including old fashioned ones that rely on traditional image processing features (e.g., canny edge, contours), and deep learning models that learn to detect from large-scale GUI data. Unfortunately, these CV methods are not originally designed with the awareness of the unique characteristics of GUIs and GUI elements and the high localization accuracy of the GUI element detection task. A detailed analysis on the unique characteristics of UI designs and a systematic empirical study on existing object detection techniques are needed to explore the potential issues and refine them.

1.2 Main Works and Contributions

This thesis consists of three works: wireframe-based UI design search engine [7], automated accessibility label generation [10], and UI element detection [36]. All works are published in top-tier conferences or journals, and all source code, models and datasets are released in GitHub repositories.

In the first work, to enable a fine-grained search of UI design and tackle the proposed challenge, we propose a deep-learning-based UI design search engine, called WAE, to fill in the gap. The key innovation of our search engine is to train a wireframe image autoencoder using a large database of real-application UI designs, without the need for labelling relevant UI designs. We implement our approach for Android UI design search and conduct extensive experiments with artificially created relevant UI designs and human evaluation of UI design search results. Our experiments confirm the superior performance of our search engine over existing image-similarity or component-matching-based methods and demonstrate the usefulness of our search engine in real-world UI design tasks

In the second work, we first conducted a large-scale empirical study on 10,408 Android mobile applications to examine the accessibility issues. We found that more than 77% of apps have issues of missing labels. Most of these issues are caused by developers' lack of awareness and knowledge in considering the minority. And even if developers want to add the labels to UI components, they may not come up with concise and clear descriptions as most of them are of no visual issues. To overcome these challenges, we develop a deep-learning-based model, called LabelDroid, to automatically predict the labels of image-based buttons by learning from large-scale commercial apps in Google Play. The experimental results show that our model can make accurate predictions and the generated labels are of higher quality than that from real Android developers.

In the third work, we perform the first systematic analysis of the problem scope and solution space of GUI element detection, and identify the key challenges to be addressed, the limitations of existing solutions, and a set of unanswered research questions. We then conduct the first large-scale empirical study of seven representative GUI element detection methods on over 50k GUI images to understand the capabilities, limitations and effective designs of these methods. This study not only sheds the light on the technical challenges to be addressed but also informs the design of new GUI element detection methods. We accordingly design a new GUI-specific old- fashioned method for non-text GUI element detection which adopts a novel top-down coarse-to-fine strategy, and incorporates it with the mature deep learning model for GUI text detection. Our evaluation on 25,000 GUI images shows that our method significantly advances the start-of-the-art performance in GUI element detection.

1.3 Thesis Statement

I believe that by understanding mobile user interfaces through semantic and data-driven analyses, the efficiency of mobile user interface development can be improved.

1.4 Thesis Outline

The following chapters are organized as follows:

Chapter 3 presents the literature review of this thesis including existing UI design datasets, UI design search engines, accessibility guidelines and empirical studies, accessibility testing tools, and UI elements detection techniques from pixel.

In Chapter 4, I design and develop a deep-learning-based UI design search engine to help designers and developers better understand the design space related to their current design, and enable them to search for similar UI designs in a targeted manner. It not only helps them create a trendy, useful and practical UI that is precise enough and contains necessary functions, it also enhances the efficiency of design prototyping.

In Chapter 5, I conduct an empirical study to evaluate the accessibility issue of lacking labels for image-based buttons to understand the accessibility support of mobile applications. I then develop a tool, LabelDroid, to assist developers in implementing the accessibility features of user interfaces.

In Chapter 6, I systematically summarize the unique characteristics of user interfaces and UI elements, and conduct an empirical study to evaluate seven state-of-the-art object detection techniques. Based on the understanding of the advantages and disadvantages of existing techniques, I develop a hybrid top-to-down coarse-to-fine element detection method, specifically designed for user interfaces.

Chapter 8 provides a summary of this thesis and proposes the potential directions in the future.

Chapter 2

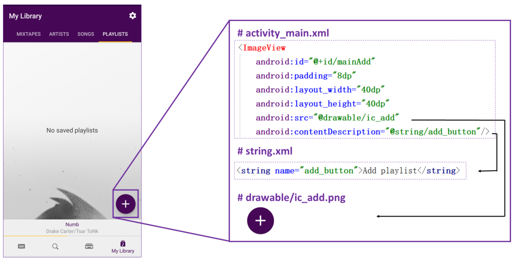
Background

In this chapter, we give some background information about Android accessibility background to assist the following reading.

2.1 Android Accessibility Background

2.1.1 Content Description of UI component

Android UI is composed of many different components to achieve the interaction between the app and the users. For each component of Android UI, there is an attribute called `android:contentDescription` in which the natural-language string can be added by the developers for illustrating the functionality of this component. This need is parallel to the need for alt-text for images on the web. To add the label "add playlist" to the button in Figure 5.2, developers usually add a reference to the *android:contentDescription* in the layout xml file, and that reference is referred to the resource file *string.xml* which saves all text used in the application. Note that the content description will not be displayed in the screen, but can only be read by the screen reader.



activity_main.xml

```
<ImageView
    android:id="@+id/mainAdd"
    android:padding="8dp"
    android:layout_width="40dp"
    android:layout_height="40dp"
    android:src="@drawable/ic_add"
    android:contentDescription="@string/add_button"/>
```

string.xml

```
<string name="add_button">Add playlist</string>
```

drawable/ic_add.png

Figure 2.1: Source code for setting up labels for "add playlist" button (which is indeed a *clickable ImageView*).

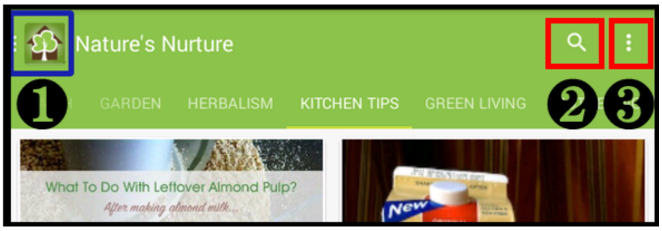

Nature's Nurture
GARDEN
HERBALISM
KITCHEN TIPS
GREEN LIVING
1
2
3
What To Do With Leftover Almond Pulp?
After making almond milk...
New

Figure 2.2: Examples of image-based buttons. ① `clickable ImageView`; ②③ `ImageButton`.

2.1.2 Screen Reader

According to the screen reader user survey by WebAIM in 2017 [37], 90.9% of respondents who were blind or visually impaired used screen readers on a smart phone. As the two largest organisations that facilitate mobile technology and the app market, Google and Apple provide the screen reader (TalkBack in Android [38] and VoiceOver in IOS [39]) for users with vision impairment to access to the mobile apps. As VoiceOver is similar to TalkBack and this work studies Android apps, we only introduce the TalkBack. TalkBack is an accessibility service for Android phones and tablets which helps blind and vision-impaired users interact with their devices more easily. It is a system application and comes pre-installed on most Android devices. By using TalkBack, a user can use gestures, such as swiping left to right, on the screen to traverse the components shown on the screen. When one component is in focus, there is an audible description given by reading text or the content description. This screen reader software adds spoken, audible and vibration feedback to your device. When you move your finger around the screen, TalkBack reacts, reading out blocks of text or telling you when you've found a button. Apart from reading the screen, TalkBack also allows you to interact with the mobile apps with different gestures. It provides spoken feedback as you navigate around the screen, by describing your actions and informing you of any notifications. For example, users can quickly return to the home page by drawing lines from bottom to top and then from right to left using fingers without the need to locate the home button. TalkBack also provides local and global context menus, which respectively enable users to define the type of next focused items (e.g., characters, headings or links) and to change global setting.

2.1.3 Android Classes of Image-Based Buttons

There are many different types of UI components [40] when developers are developing the UI such as `TextView`, `ImageView`, `EditText`, `Button`, etc. According to our observation, the image-based nature of the classes of buttons make them necessary to be added with natural-language annotations for two reasons. First, these components can be clicked i.e., as important proxies for interaction than static components like `TextView` with users. Second, the screen readers cannot directly read the image to natural language. In order to properly label an image-based button such that it interacts properly with screen readers, alternative text descriptions must be added in the button's content description field. Figure 2.2 shows two kinds of image-based buttons.

2.1.3.1 Clickable Images

Images can be rendered in an app using the Android API class `android.widget.ImageView`[41]. If the clickable property is set to true, the image functions as a button (① in Figure 2.2). We call such components Clickable Images. Different from normal images of which the clickable property is set to false, all Clickable Images are non-decorative, and a null string label can be regarded as a missing label accessibility barrier.

2.1.3.2 Image Button

Image Buttons are implemented by the Android API class `android.widget.ImageButton` [42]. This is a sub-class of the Clickable Image's `ImageView` class. As its name suggests, Image Buttons are buttons that visually present an image rather than text (②③ in Figure 2.2).

Chapter 3

Literature Review

3.1 UI Design Prototyping

UI Design Datasets: Many UI design kits [43; 44; 45; 46] are publicly available on the web. Designers also share their UI designs on social media platforms such as Dribbble [47], UI Movement [48]. They are a great source for design inspirations, but they cannot expose developers to a large UI design space of real applications. Many UI designs in these platforms are only for demonstration purposes for people to find inspirations, and some are not implemented into a real application. Furthermore, these platforms support only simple keyword-based search. Alternatively, existing applications provide a large repository of UI designs. To harness these UI designs, people resort to automatic GUI exploration methods to simulate the user interaction with GUI and collect UI screenshots of existing applications, which can support data-driven applications such as UI code generation [33; 8; 34], GUI search [18; 49; 50; 51], design mining [52], design linting [53], UI accessibility [10], user interaction modeling [32; 54] and privacy and security [55]. In the same vein, my work builds a large database of real-application UI designs by automatic GUI exploration. Different from existing work, I further wirify UI screenshots to support wireframe-based UI design search.

UI Design Search: Some techniques [49; 50; 52] support UI search by images, but they use low-level image features such as color histogram together with other UI information (if available) such as component type, text displayed. Although these image features are useful for measuring image similarities, they are agnostic of the visual semantics of a GUI. Therefore, image-wise similar UI designs are likely to be design-wise irrelevant. Other techniques [17; 18; 19] support GUI search by UI sketches. But they essentially convert both query UI and UIs in the database into a tree of GUI components and then find similar GUIs by computing the optimal matching of component trees. However, these methods are restricted to the pre-defined component matching rules, and are sensitive to the cut-off matching thresholds. Furthermore, individual component-matching heuristics often retrieve many irrelevant UI designs, because individual component matching cannot effectively encode the visual composition of components in a GUI as a whole. Different from these existing works, I propose a tool that truly models UIs as images and uses deep learning features to encode the visual semantics of UIs to retrieve design-wise relevant UIs to assist the prototyping process in a finer manner. The most related work is Rico [32], which envisions the possibility of deep learning based UI search and demonstrates several examples based on simple fully-connected layers model with highly simplified data. Compared with their work, I develop a sophisticated model suitable for the variety of real-life UI designs, implement a working prototype and conduct systematic empirical studies.

UI Implementation Automation: Nguyen and Csallner [33] detect components in UI screenshots by rule-based image processing method and generate GUI code. They support only a small set of most commonly used GUI components. More powerful deep-learning based methods [8; 56; 34] have been recently proposed to leverage the big data of automatically collected UI screenshots and corresponding code. Different from these UI code generation methods which require high-fidelity UI design image, my approach requires only UI wireframes which can be fast prototyped even for inexperienced developers. Furthermore, my method returns a set of diverse UI designs for exploring the design space, rather than the code implementing a specific UI design. Some recent works explore issues between UI designs and their implementations. Moran et al. [11] check if the implemented GUI violates the original UI design by comparing the images similarity with computer vision techniques. A follow-up work by them [57] further detects and summarizes GUI changes in evolving mobile apps. UI design search finds similar UI designs, and then these techniques may be applied to further detect the differences between similar UI designs which may help refine the search results.

3.2 App Accessibility

App Accessibility Guideline: Google and Apple are the primary organizations that facilitate mobile technology and the app marketplace by Android and IOS platforms. With the awareness of the need to create more accessible apps, both of them have released developer and designer guidelines for accessibility [58; 59] which include not only the accessibility design principles, but also the documents for using assistive technologies embedding in the operating system [59], and testing tools or suites for ensuring the app accessibility. The World Wide Web Consortium (W3C) has released their web accessibility guideline long time ago [60] And now they are working towards adapting their web accessibility guideline [60] by adding mobile characteristics into mobile platforms. Although it is highly encouraged to follow these guidelines, they are often ignored by developers. Different from these guidelines, our work is specific to users with vision impairment and predicts the label during the developing process without requiring developers to fully understand long guidelines.

App Accessibility Studies for Blind Users: Many works in Human-Computer Interaction area have explored the accessibility issues of small-scale mobile apps [61; 62] in different categories such as in health [63], smart cities [64] and government engagement [65]. Although they explore different accessibility issues, the lack of descriptions for image-based components has been commonly explicitly noted as a significant problem in these works. Park et al [62] rated the severity of errors as well as frequency, and missing labels is rated as the highest severity of ten kinds of accessibility issues. Kane et al [66] carry out a study of mobile device adoption and accessibility for people with visual and motor disabilities. Ross et al [67] examine the image-based button labeling in a relative large-scale android apps, and they specify some common labeling issues within the app. Different from their works, my study includes not only the largest-scale analysis of image-based button labeling issues to better understand the the issues, but also a solution for solving those issues by a model to predict the label of the image.

There are also some works targeting at locating and solving the accessibility issues, especially for users with vision impairment. Eler et al [68] develop an automated test generation model to dynamically spot the potential accessibility issues in the mobile apps, but fail to directly fix them. Zhang et al [69] leverage the crowd source method to annotate the GUI element without the original content description. For other accessibility issues, they further develop an approach to deploy the interaction proxies for runtime repair and enhancement of mobile application accessibility [9] without referring to the source code. Although these works can also help ensure the quality of mobile accessibility by spotting accessibility issues that need to be fixed or providing some annotation tools to ease the labeling process, they still need much effort from developers. For the missing accessibility label issues, developers still need to figure out how to add concise, easy-to-understand descriptions to the GUI components for users with vision impairment. Instead, the model proposed in my work can automatically recommend the label for image-based components and developers can directly use it or modify it for their own apps.

App Accessibility Testing Tools It is also worth mentioning some related non-academic projects. There are mainly two strategies for testing app accessibility (for users with vision impairment) such as manual testing, and automated testing with analysis tools. First, for manual testing, the developers can use the built-in screen readers (e.g., TalkBack [38] for Android, VoiceOver [39] for IOS) to interact with their Android device without seeing the screen. During that process, developers can find out if the spoken feedback for each element conveys its purpose. Similarly, the Accessibility Scanner app [23] scans the specified screen and provides suggestions to improve the accessibility of your app including content labels, clickable items, color contrast, etc. But the problem with it is that developers have to manually explore each screen and run the Accessibility Scanner on it. The shortcoming of this tool is that the developers must run it in each screen of the app to get the results. Such manual exploration of the application might not scale for larger apps or frequent testing, and developers may miss some functionalities or elements during the manual testing. However that process will be very time-consuming and labor-extensive and the developers may miss some functionalities or elements during the manual testing.

Second, developers can also automate accessibility tasks by resorting testing frameworks like Android Lint, Espresso and Robolectric, etc. The Android Lint [22] is a static tool for checking all files of an Android project, showing lint warnings for various accessibility issues including missing content descriptions and providing links to the places in the source code containing these issues. Apart from the static-analysis tools, there are also testing frameworks such as Espresso [24] and Robolectric [25] which can also check accessibility issues dynamically during the testing execution. And there are counterparts for IOS apps like Earl-Grey [70] and KIF [71]. Note that all of these tools are based on official testing framework. For example, Espresso, Robolectric and Accessibility Scanner are based on Android's Accessibility Testing Framework [72].

Although all of these tools are beneficial for the accessibility testing, there are still three problems with them. First, it requires developers' well awareness or knowledge of those tools, and understanding the necessity of accessibility testing. Second, all these testing are reactive to existing accessibility issues which may have already harmed the users of the app before issues fixed. In addition to these reactive testing, we also need a more proactive mechanism of accessibility assurance which could automatically predicts the content labeling and reminds the developers to fill them into the app. The goal of my work is to develop a proactive content labeling model which can complement the reactive testing mechanism.

3.3 UI Detection from Pixels

GUI design, implementation and testing are important software engineering tasks, to name a few, GUI code generation [8; 73; 34], GUI search [19; 74; 51; 7; 75], GUI design examination [11; 76; 53], reverse-engineering GUI dataset [32; 54], GUI accessibility [10], GUI testing [27; 29; 77; 28; 78] and GUI security [55; 79]. Many of these tasks require the detection of GUI elements. As an example, [28] shows that exploiting exact widget locations by instrumentation achieves significantly higher branch coverage than predicted locations in GUI testing, but widget detection (by YOLOv2) can interact with widgets not detected by instrumentation. My work focuses on the foundational technique to improve widget detection accuracy, which opens the door to keep the advantage of widget detection while achieving the benefits of instrumentation in downstream applications like GUI testing.

To detect the UI elements in GUIs, we need to first locate the elements and then classifies the type. Existing element detection techniques can be divided into two types: old fashioned and deep learning techniques. Old fashioned techniques rely on either edge/contour aggregation [33; 80; 34] or template matching [81; 27; 26; 30]. Nguyen et al. [33] first propose REMAUI to localise UI elements using old-fashioned techniques. They leverage Canny edge [82] and contour map [83] to detect primitive visual features, and then localise the elements based on some predefined rules. In their work, they only recognise two types of GUI elements, i.e., text elements or image elements, and two types of high-level components (container and list item). Following their work, Moran et al. [34] reuse their technique to localise GUI elements and combine the deep learning technique to classify image elements into fine-grained categories. The most recent technique Xianyu [80], proposed by Alibaba, uses image binarisation and horizontal/vertical slicing to obtain GUI elements. However, these old-fashioned techniques are error-prone to aggregate these fine-grained regions into GUI elements, especially when GUIs contain images with physical-world objects. On the other hand, template matching methods improve over edge/contour aggregation by guiding the region detection and aggregation with high-quality sample images or abstract prototypes of GUI elements. But this improvement comes with the high cost of manual feature engineering. As such, it is only applicable to simple and standard GUI widgets (e.g., button and checkbox of desktop applications). It is hard to apply template-matching method to GUI elements of mobile applications which have large variance of visual features.

In comparison, deep learning models [84; 51; 85; 28; 86] remove the need of manual feature engineering by learning GUI element features and their composition from large numbers of GUIs. Gallery D.C. [51] apply Faster RCNN [84] to detect GUI elements in UI screenshots to create a component gallery for designers and white et al. [28] leverage YOLOv2 [87] to detect GUI elements to improve GUI testing. However, while these deep learning techniques are originally designed for objects in natural scene, which has many differences from GUI elements in mobile UIs. Considering the image characteristics of GUIs and GUI elements, the high accuracy requirement of GUI-element region detection, and the design rationale of existing object detection methods, we raise a set of research questions regarding the effectiveness features and models originally designed for generic object detection on GUI elements, the region detection accuracy of statistical machine learning models, the impact of model architectures, hyperparameter settings and training data, and the appropriate ways of detecting text and non-text elements.

In our work, we aim to systematically study these research questions and we conducted the first large-scale empirical study of GUI element detection methods. Our empirical study shows that old-fashioned methods perform poorly for both text and non-text GUI element detection. Generic object detection models perform better than old-fashioned ones, but they cannot satisfy the high accuracy requirement of GUI element detection. Therefore, based on our result from empirical study, we proposed a technique, UIED, that advances the state-of-the-art in GUI element detection by effectively assembling the effective designs of existing methods and a novel GUI-specific old-fashioned region detection method.

Chapter 4

Wireframe-based UI Design Search through Image Autoencoder

4.1 Introduction

Graphical User Interface (GUI) is ubiquitous in modern desktop software, mobile applications and web applications. It provides a visual interface between a software application and its end users through which they can interact with each other. A well-designed GUI makes an application easy, practical and efficient to use, which significantly affects the success of the application and the loyalty of its users [1; 2; 3]. For example, in the competitive mobile application market, the design of an application's GUI, or even its icon, has become crucial for distinguishing an application from competitors, attracting user downloads, reducing users' complaints and retaining users [4; 5; 6].

Designing the visual composition of a GUI is an integral part of software development. Based on the initial user needs and software requirements, the designers usually first design a *wireframe* of the desired GUI by selecting highly-simplified visual components with special functions (for example those shown in Figure 4.1) and determining the layout of the selected components that can support the interactions appropriate to application data and the actions necessary to achieve the goals of users, and modify their designs iteratively by comparing with existing online design examples. They then add high-fidelity visual effects to the GUI components, such as colors and typography, and add application-specific texts and images to the GUI design. Of course, the wireframe design and the high-fidelity GUI design are interweaving and iterative during which designers continually explore the design space by removing unnecessary visual components, adding missing components, and refining the components' layout and visual effects.

To satisfy users' needs, designing a good GUI demands not only the specific knowledge of design principles and guidelines (e.g., Android Material Design [12], iOS Human Interface Guidelines [13]), but also the understanding of design space which has the great variations in visual components that can be potentially used, their layout options, and visual effect choices. As shown in Figure 4.2, the design space of a GUI, even for the simple sign-up feature, can be very large. However, due to the shortage of UI designers [88], software developers who do not have much understanding of UI design space often have to play the designer role in software development, especially in the start-up companies and open-source projects. For example, in a survey of more than 5,700 developers [14], 51% respondents reported that they do not have much UI design training but they work on UI design tasks, more so than other development tasks. In fact, when developing an application, what software developers and designers focus on are totally different. Developers try to make the application work while designers target at making it adorable [89], which makes it tough for software developers to directly work as designers. An effective mechanism is needed to support such developers to explore and learn about the UI design space in their UI design work.

Providing developers with a UI design search engine to search existing UI designs can help developers quickly build up a realistic understanding of the design space of a GUI and get inspirations from existing applications for their own application's UI design. However, compared with the well-supported code search [90; 91; 92; 93], there has been little support for UI design search. Existing UI design search methods [47; 48; 49] are based on keywords describing software features, UI design patterns or GUI components. Although keyword-based UI search could provide some initial design inspirations, a more advanced UI design search engine is still needed to explore the design space in a more targeted manner, which can directly take as input a schematic UI (e.g., a wireframe) that the developers sketch and returns high-fidelity UI designs alike to the input (see Section 4.2 for a motivating scenario). However, a few keywords can hardly describe the visual semantics of a desired UI design, such as visual components used and their layout.

As the advanced UI design search can consider UI designs as images, a naive solution could be searching UI designs by image similarity of certain image features such as color histogram [15] or Scale-Invariant Feature Transform (SIFT) [16]. Although such image features are useful for measuring image similarity, they are agonistic of the visual semantics (visual components and their compositions) of a GUI. As such, image-wise similar UI designs are very likely design-wise irrelevant. Alternatively, one can heuristically match individual visual components based on their type, position and size for measuring the similarity of two UI designs [17; 18; 19]. However, such methods are restricted by the pre-defined component matching rules, and is sensitive to the cut-off matching thresholds. Furthermore, individual component-matching heuristics often retrieve many irrelevant UI designs, because individual component matching cannot effectively encode the visual composition of components in a GUI as a whole.

The most related work is Rico [32], which introduces a new UI dataset, discusses five potential usages and demonstrates the possibility of assisting UI search. Their dataset is collected by automatic app exploration and manual exploration by recruiting crowd workers. In terms of UI search demonstration, they use a simple multilayer perceptron with only six fully connected layers, simplify the UI components as text/non-text, and show several examples without any detailed statistics on performance results. As discussed above, UI designs are sophisticated with many

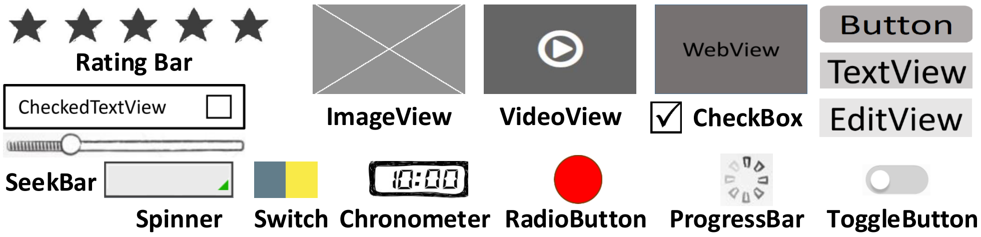


Figure 4.1: The most common wireframe components for Android UI design. Each component has its own function. For example, TextView will show text to the user and EditText enables user to input text.

variants, so it is impossible to merely use text and non-text to express the core concepts of one UI design. Besides, the complex combination of different widgets with arbitrary numbers and positions shows a huge design space in terms of typology. Therefore, a naive method with highly simplified widgets is not enough to tackle this task. Note that the Rico evaluation only shows several examples without any detailed studies on retrieval accuracy, data issue, model limitation, failure cases and usefulness evaluation. Thus, we cannot know the generalization or performance of their model.

In this chapter, I present an approach to develop a deep-learning-based UI design search engine using a convolutional neural network instead of purely fully connected layers. To expose developers to diverse, real-application UI designs for a variety of software features, I use automatic GUI exploration methods like in [8] to build a large database of UI screenshots (and their corresponding wireframes) of existing applications. I further identify 16 user interaction components which narrow the gap between designers and developers by analyzing several design platforms and UI implementation details. My approach performs wireframe-based UI design search. A wireframe captures the type and layout information of visual components, but ignores their high-fidelity visual details. As such, they can be fast prototyped and refined with minimal effort, and can retrieve visually-different but semantically relevant UI designs (see Figure 4.2 for example). My approach does not perform individual component matching, but it attempts to judge the relevance of the whole UI designs. A key challenge in developing such a robust UI-design relevance model is that no labelled relevant UI designs exist and it requires heavy manual efforts to annotate such a large dataset. Thus, I cannot use supervised learning methods like [94; 95] to train the model for encoding the visual semantics of UI designs. To overcome this challenge and relieve the heavy manual efforts, we design a wireframe autoencoder which can be trained using a large database of UI wireframes in an unsupervised way. Once trained, this autoencoder can encode both the query wireframe by the user and the UI screenshots of existing applications through their corresponding wireframes in a vector space of UI designs. In this vector space, retrieving UI screenshots alike to the query wireframe can be easily achieved by k-nearest neighbors (kNN) search.

As a proof of concept, I implement my approach for searching Android mobile application UI designs in a database of 54,987 UI screenshots from 25 categories [1] of 7,748 top-downloaded Android applications in Google Play. I evaluate the performance, generalization and usefulness of my UI design search engine[2] with an automatic evaluation of 4,500 pairs of relevant UI designs generated by component-scaling and component-removal operations, the human evaluation of the relevance of the top-10 UI designs returned for 50 unseen query UIs from 25 applications (not in my database), and a user study with 18 non-professional UI designers on five UI design tasks. My evaluation confirms the superior performance of my approach than the baselines based on low-level image features (color histogram and SIFT), individual component-matching heuristics and fully connected layers based neural network. The user study participants highly appreciate the relevance, diversity and usefulness of UI design search results by my tool in assisting their design work. They also point out several user needs for UI design search, such as constraint-aware UI design search, more flexible encoding of component layouts.

The contributions can be summarized as follows:

- We propose a novel deep-learning based approach using convolutional neural network in an unsupervised manner for building a UI design search engine that is flexible and robust in face of the great variations in UI designs.
- We build a large wireframe database of UI designs of top-downloaded Android applications by exploring different wireframing approaches, and develop a web-based search interface to implement our approach.
- Our extensive experiments demonstrate the performance, generalization and usefulness of our approach and tool support, and point out interesting future work.

4.2 Motivating Scenario

A start-up company needs to design the UIs for its mobile application. Like many small companies [88], it does not have a professional UI designer due to budget constraints. So the design work is assigned to a software developer Lucy. Lucy has some desktop software front-end development experience, but never designs a mobile application UI.

The first task for Lucy is to design a sign-up UI for collecting user information, such as user name, password and email, during user registration. Based on her prior desktop software development experience, Lucy designs a very basic sign-up UI (Figure 4.2 (a)). It has several side-by-side TextView and EditView: TextView for displaying a label for the information to be collected, and EditView for entering

[1] Since Google Play updated their app categories after my data collection, the number of categories (25) of the crawled dataset is different from the current number (35) in Google Play.

[2] All UI design images in this chapter and all experiment data and results can be downloaded at my Github repository https://github.com/chenjshnn/LabelDroid

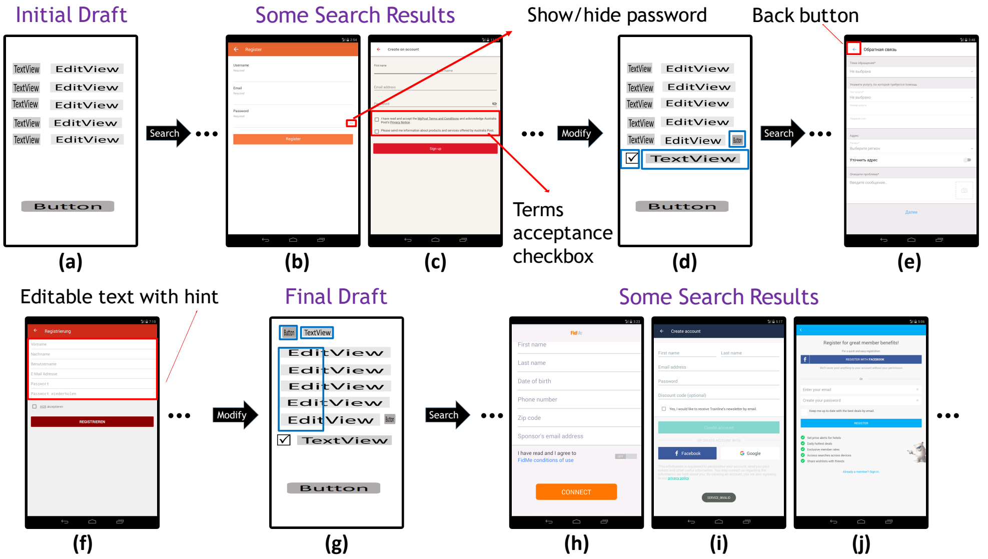


Figure 4.2: UI design search: benefits and challenge

the information. At the bottom, it has a button for submitting the entered user information.

Lucy is afraid that her design is not complete, nor trendy for mobile applications. She would like to see if other applications design sign-up UIs like hers, but she does not want to randomly download and install applications from app market just to see their sign-up UIs (if any). Not only is it time-consuming, but it also cannot give a systematic view of relevant UI designs. A better solution is to feed her UI design into an effective UI design search engine which can return similar but visual-effect-diverse UI designs from a large database of UI designs.

Lucy tries to use such a UI design search engine to obtain a list of UI designs alike to her initial sign-up UI design. Observing the returned UI designs, Lucy realizes that although her initial design has the basic functionality, it does miss some nice and important features. For example, she can add a show/hide password button (e.g., Figure 4.2 (b)), which is convenient for users to confirm the entered password. Furthermore, sign-up UI is a good place for users to access and acknowledge relevant terms and conditions (e.g., Figure 4.2 (c)). Based on such observations, Lucy refines her design as the one in Figure 4.2 (d) (changes highlighted in blue box) and search the UI design database again.

Observing the search results, Lucy gains a realistic understanding of what a trendy sign-up form needs, including visual components, layout options and visual effects, and further refines her design. For example, mobile applications often have a navigation button at the top (e.g., the back button in Figure 4.2 (b)/(c)/(e)/(f)) to facilitate the navigation among UI pages. Furthermore, unlike the traditional side-by-side label-text input design in desktop software, mobile applications use an editable text with hint to achieve the same effect (Figure 4.2 (c)/(e)/(f)). This design works better for mobile devices which have much smaller screens than desktop computer.

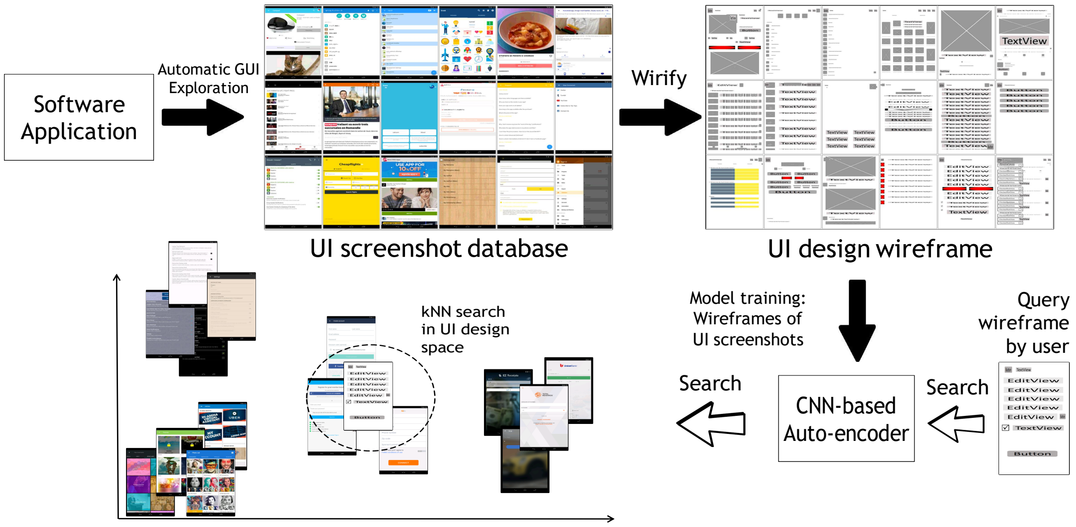


Figure 4.3: An overview of my approach. It consists of two phases: training and testing/search phases. For the training phase, I first prepare the dataset by crawling apps from Google Play and automatically exploring UIs in each app. After that, I use the collected metadata for each user interface to obtain the low-fidelity UI design, i.e. the wireframe. I then design and train a CNN-based image autoencoder model to learn the representation of each wireframe in an un-supervised manner. For the testing/search phase, given a wireframe from the designer, I first obtain the corresponding vector representation using the trained model, and then use kNN to find the similar designs.

Based on these design inspirations, Lucy further refines her design as the one in Figure 4.2 (g). Comparing the UI design search results ((e.g., Figure 4.2 (h)/(i)/(j)) with the design in Figure 4.2 (g), Lucy is now confident in her final UI wireframe. Furthermore, observing many relevant and diverse UI designs gives Lucy many inspirations for designing high-fidelity visual effects (e.g., color system, typography) for her UIs.

As UI designs in Figure 4.2 shows, the design space of a GUI can be very huge, with the great variations in: (1) the type of visual component used (e.g., checkbox in many UI designs versus switch unique in Figure 4.2 (h)); (2) the number of visual components in a design (e.g., editable text and button in Figure 4.2 (g), (i) and (j)); (3) the position and size of visual components (e.g., editable text and button in Figure 4.2 (b) versus (h)); and (4) the layout of visual components (e.g., side-by-side label-textinput in Figure 4.2 (a) versus up-down label-textinput in Figure 4.2 (b), or left-checkbox + right-text in Figure 4.2 (c) versus left-text + right-switch in Figure 4.2 (h)). Achieving the above-envisioned benefits of UI design search requires the search engine to be flexible and robust in face of the great variations, and to achieve a good balance between similarity and variation in UI designs.

4.3 Approach

Figure 4.3 presents the overview of my proposed deep-learning based approach for building a UI design search engine that is flexible and robust in the face of the great variations in UI designs. My approach consists of three main steps: 1) build a large database of diverse, real-application UI designs using automatic GUI exploration based methods (Section 4.3.1); 2) train a CNN-based wireframe autoencoder for encoding the visual semantics of UI designs using a large database of UI design wireframes (Section 4.3.2); and 3) embed the UI designs in a latent vector space using the trained wireframe encoder and support wireframe-based kNN UI design search (Section 4.3.3)

4.3.1 Large Database of Real-Application UI Designs

A large database of diverse, real-application UI designs for a variety of different software features is necessary to expose developers to the realistic UI design space. To that end, I adopt automatic data collection method to first build a large database of UI designs from existing applications, and then use collected data to further construct my wireframe dataset.

4.3.1.1 Automatic GUI Exploration

Different techniques can be used for automatically explore the GUIs of mobile applications [8; 32], web applications [52; 50], or desktop applications [96]. Although technical details are different, these techniques work conceptually in the same way. They automatically explore the GUI of an application by simulating user interactions with the application, and output the GUI screenshot images and the runtime visual component information which identifies each component's type and coordinates in the screenshots. During the GUI exploration process, the same GUIs may be repeatedly visited, but the duplicated screenshots are discarded to ensure the diversity of the collected UI designs. To enhance the quality of the collected UI designs, further heuristics can be implemented to filter out meaningless UIs, for example, the home screen of mobile device, the simple UIs without much design like a UI with only one image component. In detail, I first crawl apps from Google Play, and automatically install and run the app in the simulator. For each app, my simulator interacts with the app by simulating the user's actions including clicking buttons, entering text, and scrolling the screen. When entering one new page, my tool will take a screenshot of the current UI and dump the XML runtime code. The XML runtime code contains all information about the current UI, including all contained components with their corresponding bounds, class, text, boolean attributes regarding executability (such as checkable, clickable and scrollable) and the hierarchical relationship among them. To ensure the coverage of the explored UIs, I also apply the rules set in [8], which define the probability (or weight) of each potential actionable component to be pressed. There rules are defined as: (1) actions with higher frequency are given lower weights since I need to give other rare actions chance to

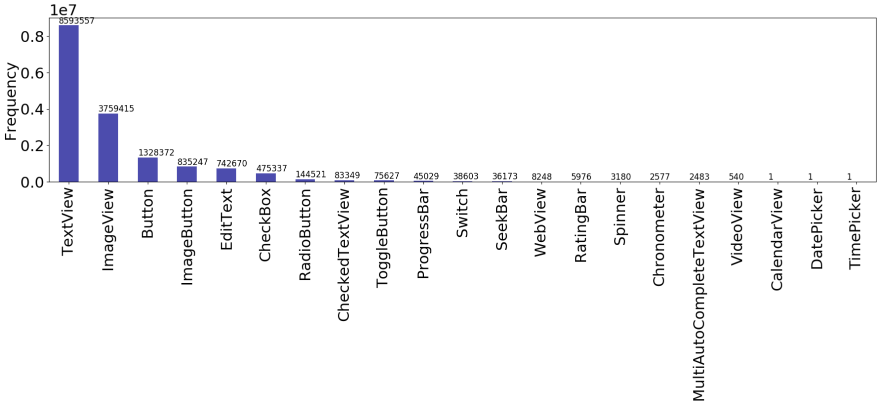


Figure 4.4: Bar chart of the frequency of each component contained in our dataset

perform; (2) actions which would lead to more subsequent UIs would have higher weights in order to explore various UIs; and (3) some special actions (such as *hardware back* and *scroll*) would be controlled in case they close current page or impact others' actions at the wrong time. The actual weights of each executable components are given by $weights(a) = (\alpha * T_a + \beta * C_\alpha)/\gamma * F_a$, where a, T_a, C_α are the action, the weights of different types of actions and the number of unexplored executable components in current UI respectively, and α, β, γ are the hyperparameters. Since this collection process is automatic, some UIs may be revisited several times and I need to remove duplicate data. To this end, I compare current dumped XML code files with the collected data by comparing the hash value of GUI component sequences.

4.3.1.2 Wirification

My approach performs wireframe-based UI design search. Therefore, different from existing reverse-engineering methods, I need to further obtain a UI wireframe for each collected UI screenshot in the database. To that end, I have two steps. First, I define a set of wireframe components that are essential for different kinds of user interactions at the design level by analysing popular designer's tools and the underlying implementation details of UIs. Second, I find the "right" representation of each component by my exploration experiments.

Selection of component types. First, there are two types of Android UI components in terms of functionality: layout components and UI control components [97]. The UI control components (e.g., button, textView, ImageView) are the visible components I could see and interact with, while layout components (e.g., linearLayout, relativeLayout) are used for constraining the position relationship among UI control components. As the input of this work is just the wireframe which involves more about the control component selection with rough position, I am concerned with only UI control components (I briefly mention it as UI components). There are 21 UI components in our dataset, which I choose as the candidate components for the wireframe. When converting an UI screenshot with corresponding run-time code,

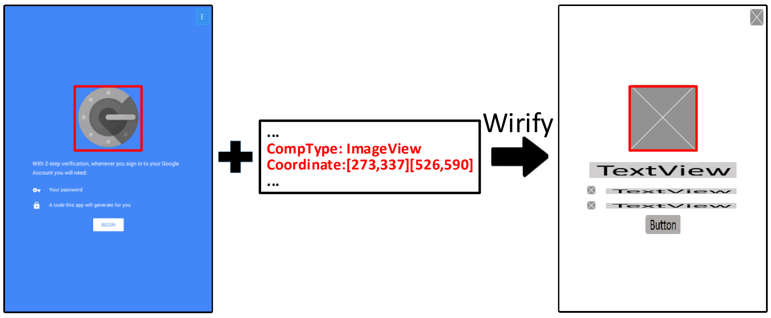


Figure 4.5: The visual rendering of the wireframe illustrates that top large ImageView. The view hierarchy information contains the type and coordinates of each element contained in the user interface. We use this information to generate the corresponding wireframe for each UI.

I consider two factors: (1) Components with similar function and similar visual effect would not have much difference when designing the wireframe; (2) Components which are rarely used may not be very useful for the UI design. For the first consideration, I merge MultiAutoCompleteTextView with EditText. Both of them enable editable text, but MultiAutoCompleteTextView has additional text auto-complete function. However, this function can be achieved in EditText by manifesting the underlying background code. I also merge ImageButton with Button because they both enable users to click them and then trigger some events. In terms of the second consideration, I ignore CalenderView, TimePicker and DatePicker components as they appear only once in our dataset(see Figure 4.4). The low frequency may because they further separate into several children components, such as TextView and Spinner. As a result, I leave with 16 components as my final set of wireframe units. The 16 types of components are also widely covered by popular wireframe tools for mobile UI design like Adobe XD [98], Fluid UI [99], Balsamiq Mockups [100]. In the implementation of the wireframing process, I use the representation of EditText to represent MultiAutoCompleteTextView, and the representation of Button to represent ImageButton in the wireframe. I do not draw CalenderView, TimePicker and DatePicker in the wireframe for the above reason. For other components, I draw them with their own representations in the wireframe. I release the source code of the wireframe transformation in my Website[3].

Wirification Process. After defining the 16 core wireframe components, a UI screenshot is then wirified into a UI wireframe using the XML runtime code file I dumped during the automatic explorations of apps. Note that there is no uncertainty during this process as it is completely a rule-based process. I wireframe the screenshot according to its dumped run-time code directly from the Android operating system, which contains the type and coordinates of each component in a UI screenshot. Therefore, these UI screenshots and the corresponding runtime code files are perfectly matched, and there will be no error during the wireframe transforma-

[3] https://github.com/chenjshnn/LabelDroid

tion. Figure 4.5 illustrates this high-level wirification process: the UI wireframe is of the same size as the UI screenshot, and has a white canvas on which a wireframe component is drawn at the same position and of the same size as each corresponding visual component in the UI screenshot (e.g., ImageView). However, the wireframe components ignore the color and the text/image content of the corresponding visual components.

Exploration of the best representation way of wireframes. In addition to this, I need to define the representation of these components to construct my final wireframe dataset. I do not use the default images of popular tools [98; 100; 99] as they are not precise or general enough. Instead, I represent them with simple rectangles in different colors, which can explicitly tell the model that those components are different. Due to the huge design space, it is unrealistically to consider all colors and color palettes, so I consider three typical variants to represent these visual components, namely different grey-scale values, different colors, and different colors with different textures. The detailed exploration setup and results of the best representation of components will be discussed later in Section 4.5. Note that to avoid potential distraction, we present the wireframe as text with different grey-scale color background to help readers better understand these wireframes in the chapter.

4.3.2 CNN-Based Wireframe Autoencoder

Determining the relevance of UI designs is a challenging task, in that it requires encoding not only visual components individually, but also the visual composition of the components in a UI as a whole. The design space of what components to use and how to compose them in a UI is huge, and thus cannot be heuristically enumerated. CNN-based model can automatically learn latent features from a large image database, which has outperformed hand-crafted features in many computer vision tasks [101; 95; 102]. Although I have a large database of UI designs, the relevance of these UI designs are unknown. Therefore, I have to train a CNN model for encoding the visual semantics of UI designs in an unsupervised way. To that end, I choose to use a CNN-based image autoencoder architecture [103] that requires only a set of unlabeled input images for model training. As illustrated in Figure 4.6, our autoencoder takes as input a UI wireframe image. It has two components: an encoder compresses the input wireframe into a latent vector representation through convolution and downsampling layers, and then a decoder reconstructs an output image from this latent vector representation through upsampling and transposed convolution layers. The reconstructed output image should be as similar to the input image as possible, which indicates that the latent vector captures informative features from the input wireframe design for reconstructing it. This latent vector representation of UI designs can then be used to measure the relevance of UI designs.

Convolution A convolution operation performs a linear transformation over an image such that different image features become salient. According to the research of CNN visualization [104; 101], shallow convolutional layers detects simple features such as edges, colors and shapes, which are then composed in the deep convolutional

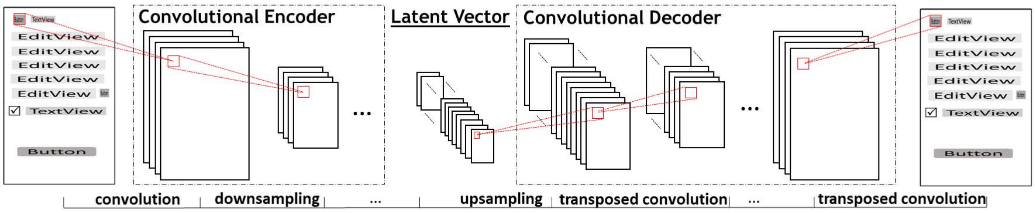


Figure 4.6: The architecture of our wireframe autoencoder

layers to detect domain-specific features (e.g., the visual semantics of UI designs in our work).

An image is represented as a matrix of pixel values, i.e., $0 \leqslant p_{hwd} \leqslant 255$ where h, w and d are the height, width and depth of the image. $d = 1$ for grayscale image and $d = 3$ for RGB color image. The convolution of an image uses a *kernel* (i.e., a small matrix like $3 \times 3 \times d$ of learnable parameters) and slide the kernel over the image's height and width by 1 pixel at a time. At each position, the convolution operation multiplies the kernel element-wise with the kernel-size subregion of the image, and sums up the values into an output value. The transposed convolution is the opposite to the normal convolution. It multiples a value with a kernel and outputs a kernel-size matrix. A convolutional layer can apply a number of kernels (n). The output matrix ($h \times w \times n$) after a convolutional layer is called a feature map, which can be fed into the subsequent network layers for further processing. Each kernel map $h \times w$ in the feature map corresponds to a kernel, and can be regarded as an image with some specific features highlighted.

Downsampling & Upsampling Within the encoder, downsampling (also called pooling) layers take as input the output feature map of the preceding convolutional layers and produce a spatially (height and width) reduced feature map. A downsampling layer consists of a grid of pooling units, each summarizing a region of size $z \times z$ of the input kernel map. As the downsampling layer operates independently on each input kernel map, the depth of the output feature map remains the same as that of the input feature map. In our architecture, I adopt *1-max pooling* [105] which takes the maximum value (i.e., the most salient feature) in the $z \times z$ region. 1-max pooling brings the benefits of the invariance to image shifting, rotation and scaling, leading to a certain level of insensitivity to encoding component spatial variations in UI designs.

Within the decoder, I use the upsampling layers which are opposite to downsampling. They increase the spatial size (height and width) of the feature map by replacing each value in the input feature map with multiple values. In our architecture, I adopt the nearest-neighbor interpolation [106], i.e., enrich the original pixel in the feature map into a $z \times z$ region with the same value as the original pixel. The upsampling layers progressively increase the spatial size of the feature map until the decoder finally reconstructs an output wireframe of the same size as the input wireframe.

Model Training The encoder and the decoder are trained as an end-to-end sys-

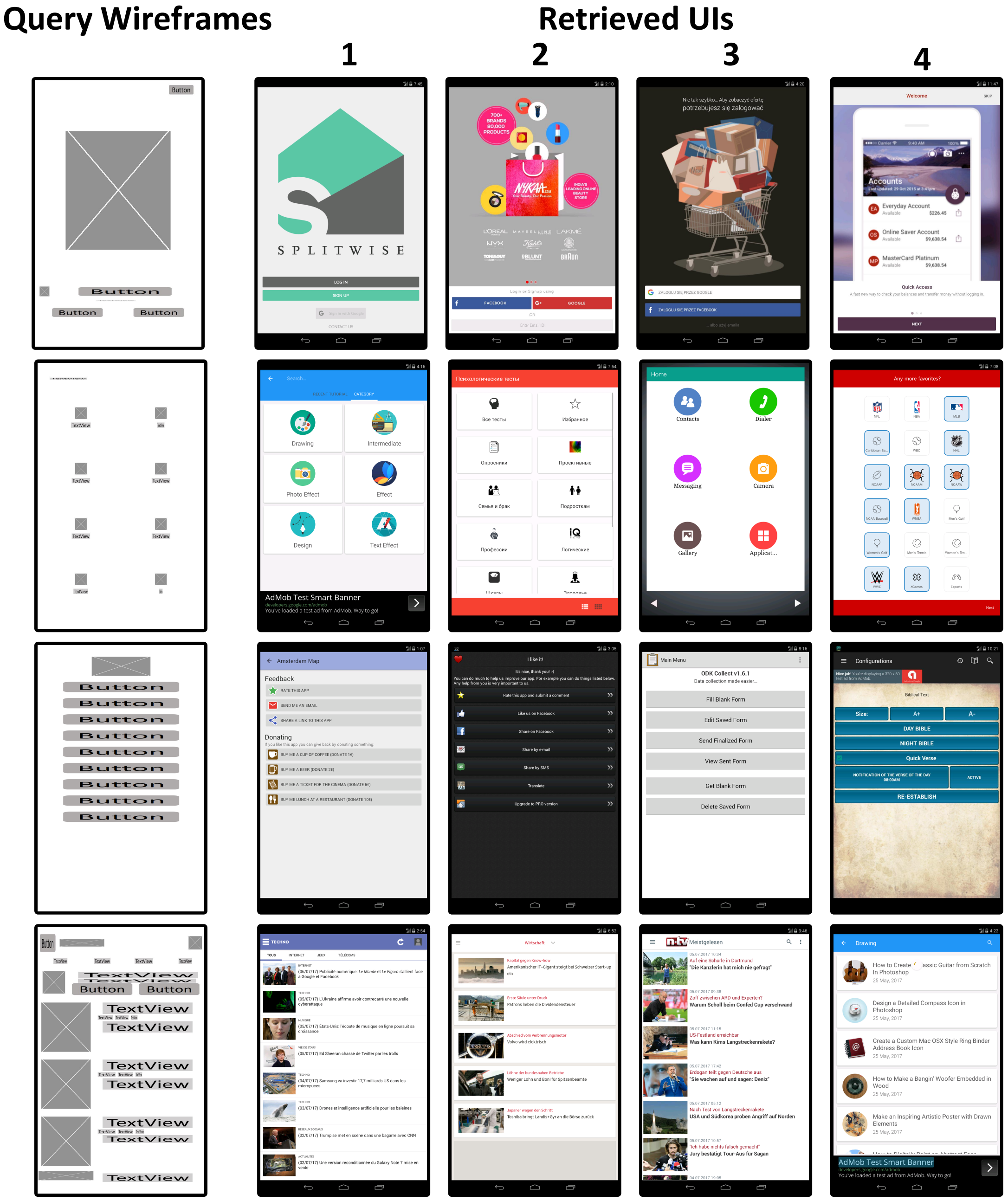


Figure 4.7: Examples of kNN search in UI design space

tem. Given a UI wireframe image X, the encoder compresses it into the latent vector V: $\phi : X \rightarrow V$ where ϕ represents the function of the encoder's convolutional and downsampling layers. Then the decoder decodes the latent vector V into an output wireframe image Y: $\psi : V \rightarrow Y$ where ψ represents the function of the decoder's upsampling and transposed convolutional layers. The target is to minimize the difference between the input wireframe X and the output wireframe Y: $argmin_{\phi,\psi} \|X - Y\|^2$. I train the wireframe autoencoder to minimize the reconstruction errors with mean square error (MSE) [107], i.e., $\mathcal{L}(X, Y) = \|X - Y\|^2$. At the training time, I optimize the MSE loss over the training dataset using stochastic gradient descent [108]. The decoder backpropogates error differentials to its input, i.e., the encoder, allowing us to train a wireframe encoder using unlabelled input wireframes.

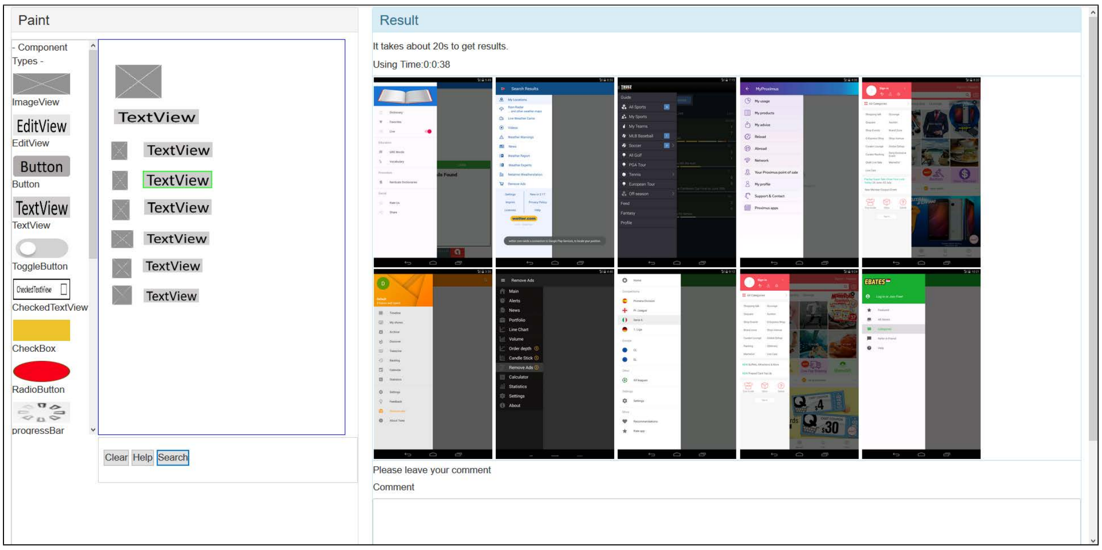
4.3.3 kNN Search in UI Design Space

As shown in Figure 4.2, by training the wireframe autoencoder, I obtain a convolutional encoder which can encode an input wireframe into a latent vector representation. Given a database of automatically-collected UI screenshots (can be different from the UI screenshots used for model training), I use this trained wireframe encoder to embed the UI screenshots through their corresponding wireframes into a UI design space S. Each UI screenshot uis is represented as a latent vector $V(uis)$ in this UI design space. Given a query wireframe wf_q drawn by the user, I also use the trained wireframe encoder to embed wf_q into a vector $V(wf_q)$ in the UI design space. Then, I perform k-nearest neighbors (kNN) search in the UI design space to find the UI screenshots uis whose embedding is the top-k most similar (by Mean Square Error (MSE) in this work) to that of the query wireframe, i.e., $argmin^k_{uis \in S} ||V(uis), V(wf_q)||^2$. Figure 4.7 shows some examples of UI design results from my empirical studies. The fourth example shows that my model can successfully encode the visual semantics of rather complex UI designs.

4.4 Implementation

4.4.1 Data Collection

We develop a proof-of-concept tool for searching Android mobile application UI designs. The backend UI design space contains 54,987 UI screenshots from 7,748 Android applications belonging to 25 application categories. We crawl the top-downloaded Android applications from Google Play, because studies show that the download number of an application correlates positively with the quality of the application's GUI design [4; 5]. There are three types of Android Apps: native, hybrid and web apps [109]. The underlying implementations of these types are different. Native apps use Android native widgets or widgets derived from them, hybrid apps utilize WebView to encode their HTML/CSS part components into an Android Application, and web apps directly use HTML/CSS/JavaScript. In this chapter, we only



It takes about 20s to get results.

Using Time:0:0:38

Please leave your comment

Figure 4.8: The core page of our User Interface Search Website.

collect UIs from native and hybrid applications because they are easy to download and install from app store, while there is no such "app store" for web applications. We remove some UIs, whose WebView takes over half of the screen. We keep small WebView component because most of them are advertisement

We use the automatic GUI exploration method in [8] to build a large database of UI screenshots from these Android applications, and the detailed process is stated in Section 4.3.1.1. In total, we crawled 8,000 Android apps from Google Play with the highest installation numbers and successfully ran 7,748 Android applications and collected 54,987 UI screenshots. Note that some apps were discarded due to the need of extra hardware support or the absence of some certain third party libraries in our emulator. The median number of UI screenshots per application is three. Our database contains very diverse UI designs (see Figure 4.3 for some randomly selected examples). More examples can be seen in our Github repository[4].

4.4.2 Model Hyperparameters

The wireframe autoencoder in our tool is configured as follows. The input wireframe is a RGB color image and scaled to 180 × 228 for efficient processing. The encoder uses four convolutional layers, which use 16 3x3x3 kernels, 32 3x3x16 kernels, 32 3x3x32 kernels, and 64 3x3x32 kernels, respectively. Each convolutional layer is followed by a $ReLU(x) = max(0, x)$ non-linear activation function, a 1-max pooling layer with 2 × 2 pooling region, and a batch normalization layer [110]. The decoder upsamples a value in the input kernel map into a 2 × 2 region of that value. It has four upsampling layers. After each upsampling layer, the decoder uses a transposed convolutional layer, which uses 32 3x3x64 kernels, 32 3x3x32 kernels, 16 3x3x32 ker-

[4] https://github.com/chenjshnn/WAE

nels, 3 3x3x16 kernels, respectively.

4.4.3 Tool Implementation

We use k=10 for KNN in all our experiments and our tool[5]. Figure 4.8 shows the frontend of our tool. A demo video of this search interface is available in our Github repository[4], which demonstrates the UI design search process of our motivating scenario. Using our tool, the user draws a UI wireframe on the left canvas. The tool currently supports 16 most frequently-used types of wireframe components as shown in Figure 4.1. We identify these wireframe components as core for Android mobile applications by surveying Android GUI framework and popular UI design tools such as Adobe XD [98], Fluid UI [99], Balsamiq Mockups [100] as stated in Section 4.3.1.2. Once the user clicks search button, the system returns the top-10 (i.e., k=10 for KNN) UI designs in the UI design space that are most similar to the wireframe on the drawing canvas. The user can iteratively refine the wireframe and search relevant UI designs.

4.5 Effective of the different representation of wireframes

In this section, we evaluate the effectiveness of the different representation of wireframes and try to answer these questions: Which kind of color palates used to represent the wireframe performs the best? Why does the performances differ?

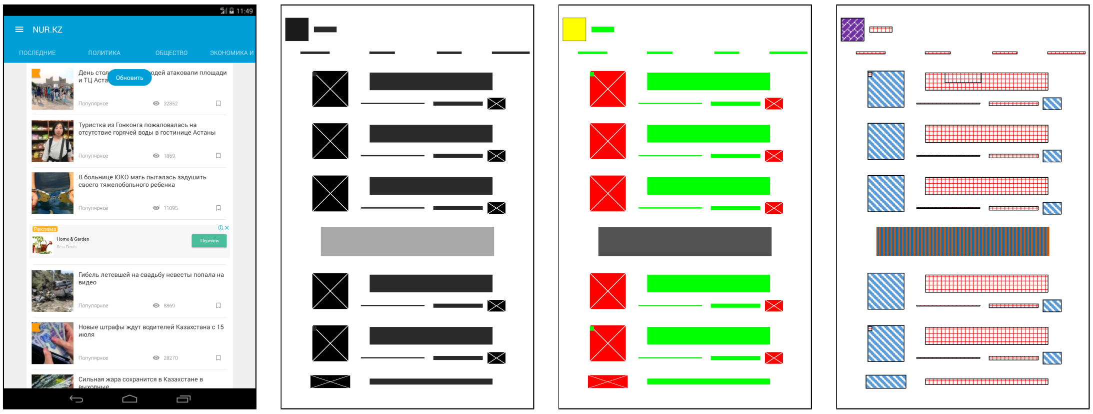
4.5.1 Dataset

To answer these questions, we first construct several wireframe datasets using different representations of wireframes as the training datasets. In details, we investigate three types of representation of visual components, including different grey-scale values, different colors and different colors with different textures. We denote these as grey-level, color-level and texture-level wireframes respectively. An example of these three representations can be seen in Figure 4.9. For three kinds of training dataset, we can directly generate them using the method stated in Section 4.3.1.2.

Second, to evaluate the performance of a UI-design search method in terms of different representations of wireframe, we require a dataset of relevant UI designs. Unfortunately, no such datasets exist. It is also impossible to manually annotate such a dataset in a large UI design database for large-scale experiments of a method's capability in face of different UI design variations. Inspired by the data augmentation methods used for enhancing the training of deep learning models [116; 117], we change a UI screenshot in our Android UI design database to artificially create pairs of relevant but variant UI designs.

Based on the position/size of components in a UI screenshot (see Section 4.3.1), we perform two types of change operations which are suitable for UI designs: *com-*

[5] We do not make K too big as developers tend not to browse a long list of recommendations [111; 112; 113; 114; 115].



День стол… …одей атаковали площади и ТЦ Аста…

Туристка из Гонконга пожаловалась на отсутствие горячей воды в гостинице Астаны

В больнице ЮКО мать пыталась задушить своего тяжелобольного ребенка

Гибель летевшей на свадьбу невесты попала на видео

Новые штрафы ждут водителей Казахстана с 15 июля

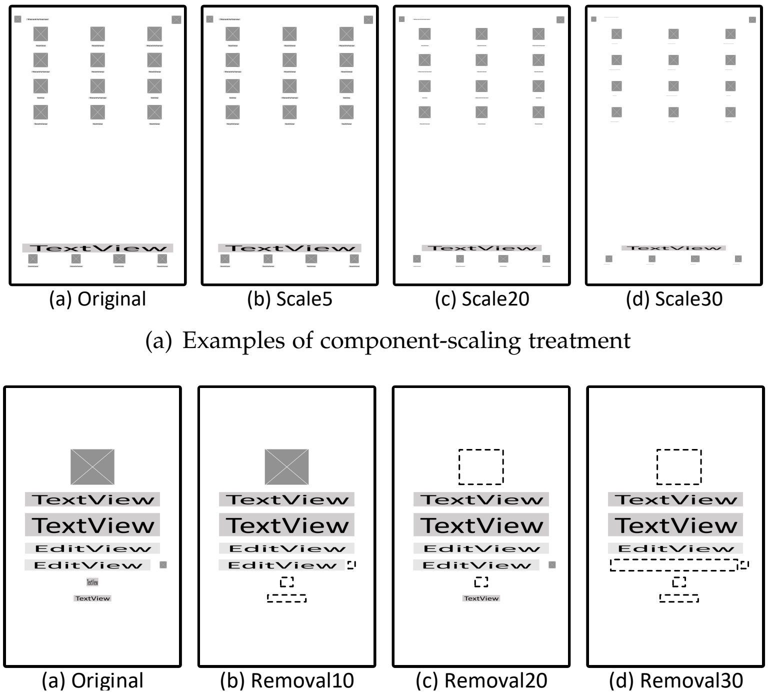
Сильная жара сохранится в Казахстане в выходные

(a) Original (b) Grey-level (c) Color-level (d) Texture-level

Figure 4.9: Example of different input format. From left to right, they are original UI images, color-level wireframe, grey-level wireframe and texture-level wireframe

ponent scaling and *component removal*. Component scaling is to scale down all visual components in a UI wireframe to their center point by 5%, 10%, 15%, 20%, 25%, or 30% pixels (round-up) of their original size (see Figure 4.10(a)) which simulates design variations in component position/size. Component removal is to randomly remove some visual components that cover 10%±5 %, 20%±5% or 30%±5% of the total area of all components (see Figure 4.10(b)) which simulates design variations in component type/number. We denote 10%±5 %, 20%±5% or 30%±5% in component removal treatment as removal10, removal20 and removal30 for simplicity. As the examples in Figure 4.2 show, such design variations are commonly present in relevant UI designs. In reality, these two types of design variations may occur at the same time. But we perform the two types of changes separately to investigate a search method's capability of handling different types of design variations.

We randomly select two sets of screenshots from 25 categories, and each set is comprised of 500 screenshots. Note that for each set, the proportion of the screenshots taken from each category is the same as the original proportion of each category in the total database. We then apply the six scaling treatments to the first set and the three component-removal treatments to the second set. The UI screenshots in the second set should have at least 5 UI components so that there are some components left after component removal. As a result, we obtain 4500 pairs of original-treated UIs, which are considered as relevant but variant UI designs. We have nine experiments (one for each treatment). For example, the Scale10 experiment uses the UIs obtained by 10% component scaling as query. Note that we generate the corresponding experimental wireframe dataset three times using the above mentioned three types of representation of wireframes, which means that we have three experimental datasets, each of them contains 4500 pairs of original-treated UIs. Using this dataset, we evaluate how well a search method can retrieve the original UI in the database using a treated UI design as query in terms of different representation of wireframes.



(a) Examples of component-scaling treatment

(b) Examples of component-removal treatment

Figure 4.10: Examples of component-scaling and component-removal treatment

4.5.2 Evaluation Metrics

We evaluate the performance of a UI-design search method by two metrics: Precision@k (Pre@k) (k=1) and Mean Reciprocal Rank (MRR). The higher value a metric is, the better a search method performs. Precision@k is the proportion of the top-k results for a query UI that are relevant UI designs. As we consider the original UI as the only relevant UI for a treated UI in this study, we use the strictest metric Pre@1: Pre@1=1 if the first returned UI is the original UI, otherwise Pre@1=0. MRR computes the mean of the reciprocal rank (i.e., 1/r) of the first relevant UI design in the search results over all query UIs.

4.5.3 Results

Quantitative Results. Figure 4.11 shows the results of evaluating the three different representation wireframes. Overall, the color-level model remains a slight advantage over grey-level model in both component-scaling and component-removal treatments, with a 5%-10% and 0.01-0.1 increase in Pre@1 and MRR respectively. The reason may be that the color-level model mainly focuses on the boundary of contained components in a wireframe instead of the exact pixel values, and the color-level wireframe input includes three channels encoding more information while the grey-level wireframes includes only one channel. In contrast, there are large margins between the performances of the color-level model and of the texture-level model, especially

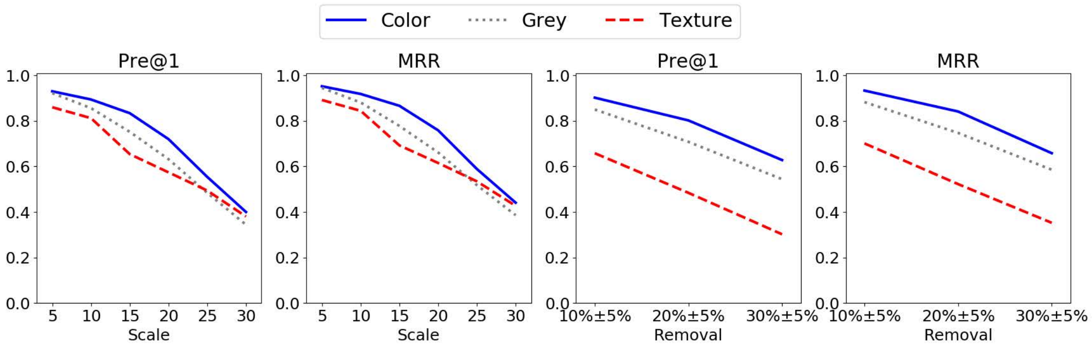


Figure 4.11: Results of three types of representation

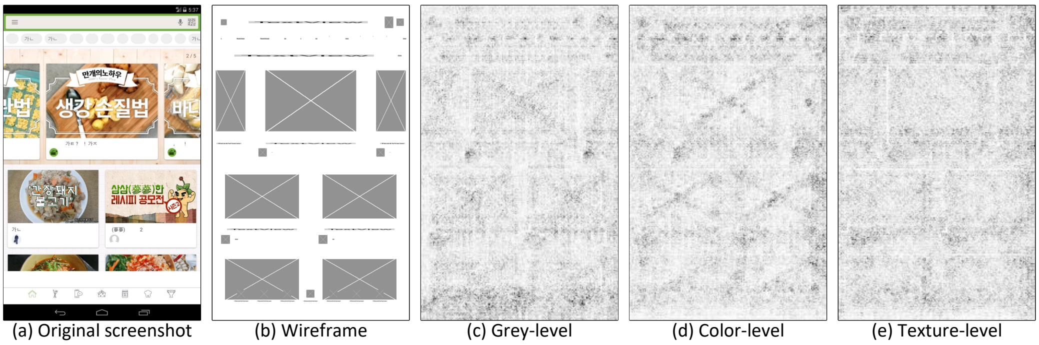


Figure 4.12: Heatmaps for the grey-level, color-level and texture-level wireframe

in component-removal treatments. This may be because the texture information is too complex and may confuse the model after several max-pooling layers.

CNN Visualization. To better understand the impacts of different representations, we visualize these models using vanilla (i.e., standard) backpropagation saliency [118] in Figure 4.12. We can find that the heatmap of the color-level model is the clearest, while that of the texture-level model is the vaguest with much noise. The grey-level heatmap is vaguer than color-level one because the differences between component and background is small in the grey one. In conclusion, the color-level model performs the best and we choose it as the representation of our wireframe dataset.

4.6 Accuracy Evaluation

Our deep-learning based approach for UI design search is the first technique of its kind. It is designed to find relevant UI designs in face of the great variations in UI designs. In this evaluation, our goal is to evaluate how well our approach achieves this design goal, and how well it compares with image-similarity based or component-matching based UI design search. We use the color-level wireframe dataset in this evaluation as the effectiveness of this kind of representation have been proved in Section 4.5.3.

4.6.1 Dataset and Metrics

To automatically evaluate and compare the effectiveness of our model and the baselines, we use the same treatments to construct the experimental dataset, while at this time, we only need to consider one kind of representation of wireframes. We again randomly select 500 UI images from 25 categories, which are different from the data in Section 4.5, to construct the experimental dataset, and then apply the nine treatments stated in Section 4.5.1. In total, we have 4500 pairs of original-treated UIs, which are considered as relevant but variant UI designs. Using this dataset, we evaluate how well a search method can retrieve the original UI in the database using a treated UI design as query. Beside, we also take the same metrics stated in Section 4.5.2

4.6.2 Baselines

We consider four baselines: two of them computes image similarity using simple color histogram and advanced SIFT feature respectively, the third one implements the component-matching heuristics proposed by GUIFetch [18], and the last one uses the naive neural network with fully connected layers from Rico [32].

Image-feature based similarity. Color histogram is a simple image feature that represents the distribution of colors in each RGB (red, green, blue) channel. It has been widely used for image indexing and image retrieval [15; 119; 120]. The scale-invariant feature transform (SIFT) [16] is an advanced image feature widely used for image retrieval [121; 122], object recognition [16], image stitching [123]. It locates keypoints in images and use the local features of the keypoints to represent images. Different images can have different numbers of keypoints but each keypoint is represented in a same-dimensional feature vector. These two baselines return the top-k most similar UI designs by the image-feature similarity.

Heuristic-based component matching. GUIFetch [18] is a recently proposed technique for searching similar GUIs by a similarity metric computed from the matched components between the two GUIs. It matches the components of the same type. The similarity of the two components is calculated based on the differences of the two components' x-coordinate, y-coordinate, length and width. If the difference of one factor is within a given threshold, the similarity score increases by 10, otherwise 0. After computing the similarity score for each pair of components in the two UIs, it uses a bipartite matching algorithm [124] to determine an optimal component matching. The similarity scores of the matched components are summed up and then divided by the maximum similarity value that the components in the query UI can have (i.e., 40 multiplies the number of components in the query UI). It then returns the top-k UIs with the highest similarity scores to the query UI.

Neural-network-based matching. Rico [32] is a UI dataset introduced to support various tasks in the UI design domain. It demonstrates the potential usage of UI search based on a naive neural network with six fully connected layers within the autoencoder framework. The latent vectors from their model are used as the features of their wireframe dataset. In terms of inference, they first extract the latent vector of the query wireframe, compare it with the latent vectors of their dataset, and then return the nearest neighbors as recommendations. For a fair comparison, we adopt the same configuration mentioned in their paper for training the model on our dataset.

4.6.3 Results

Runtime Performance We run the experiments on a machine with Intel i7-7800X CPU, 64G RAM and NVIDIA GeForce GTX 1080 Ti GPU. Take the inference time of the Scale10 experiment as an example. Our W-AE (short for Wireframe Autoencoder), Rico, GUIFetch, SIFT and color-histogram take 561.2 seconds, 771.4 seconds, 7446.9 seconds, 3944.6 seconds and 523.6 seconds for 500 queries, respectively. In general, W-AE is about 12 times and six times faster than the GUIFetch and SIFT baselines, and is as fast as the color-histogram and Rico baselines.

Retrieval Performance Figure 4.13 shows the performance metrics of the five methods in the nine treated-UI-as-query experiments. The color-histogram baseline and the SIFT baseline have close performance in all component-scaling experiments. At the component scaling ratio 10%, their performance metrics become lower than 0.2, and at the ratio 20% or higher, their performance metrics become close to 0. For component removal experiments, the advanced SIFT feature performs better than the simple color histogram feature. This is because the UIs treated by component removal still have many intact components (see Figure 4.10(b)), which have the same keypoints (and thus the same SIFT features) as their counterparts in the original UIs. This helps to retrieve the original UIs for the component-removal-treated UIs. Nevertheless, at the component-removal ratio 20% or higher, the performance metrics of the SIFT baseline become lower than 0.5.

In contrast, our W-AE is much more robust in face of large component-scaling and component-removal variations, because our CNN model can extract more abstract, sophisticated UI-design related image features through deep neural network, which are much less sensitive to image differences than low-level image features like color histogram or SIFT. At the component scaling ratio 20%, our W-AE still achieves 70.0% Precision@1 and 0.73 MRR. The performance of our W-AE degrades (but is still much better than the four baselines) when the component-scaling ratio is 25% or higher. This is because many small-size visual components (such as checkbox, switch or small text) will become hardly visible even for human eyes (see Figure 4.10(a)). Similarly, the features of such extremely-small components will become invisible to the "eye" (i.e., convolutional kernels) of the CNN model, and thus cannot contribute to the measurement of the UI-design similarity. Although such extremely-small UI components can test the limits of a search method in extreme conditions, they would rarely exist in real-word UIs because they are not user friendly. At the component-removal ratio 20%, our W-AE still achieves 84.6% Precision@1 and 0.88

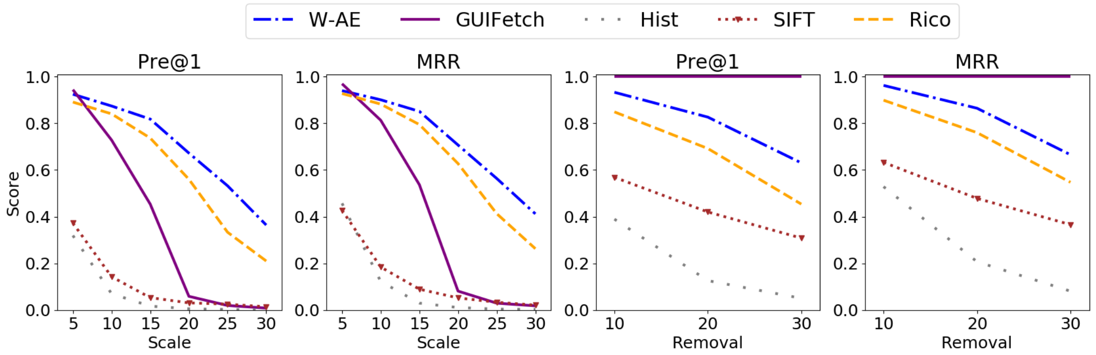
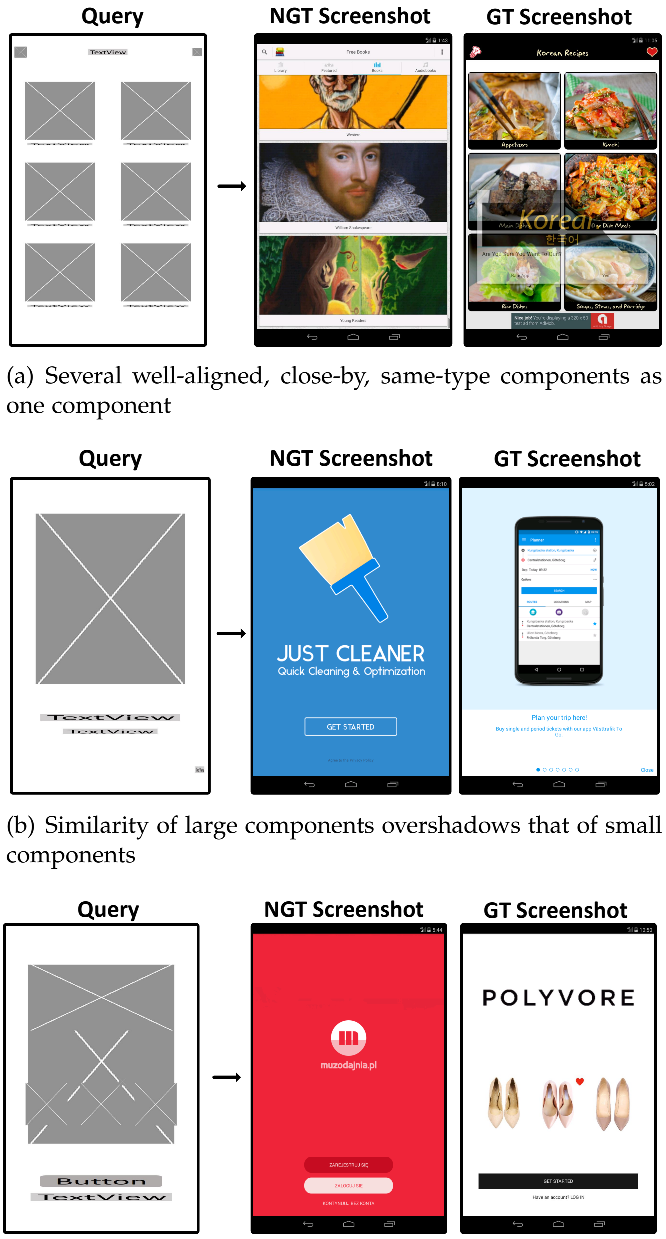


Figure 4.13: Results of nine automatic experiments

MRR. The model performance degrades (again still much better than the baselines) at the component-removal ratio 30%. However, as the example in Figure 4.10(b) shows, the treated UI with components covering 30% less area than the original UI may become not-so-similar anymore to the original UI. But we still consider the original UI as the ground truth for the treated UI in our automatic experiments, which may result in the biased metrics for all the evaluated methods.

Our W-AE outperforms Rico on all metrics by large margins. The Rico baseline performs better than other baselines but the performance gap between Rico and our W-AE model keeps growing as the UI components and layout similarity decreases. Within the component-scaling experiments, the Rico baseline is comparable to our W-AE at the scaling ratio 5%-10%, but degrades quickly when the ratio is 25% or higher. This is because Rico applies fully connected layers, which consider every pixel in the UIs without filtering out meaningless and noisy ones, leading to sensitivity to small input changes. In comparison, our W-AE performs convolution and pooling strategies to extract the core features from UIs which is much more stable. A similar observation also applies to experiments of removal treatment. The fully connected neural network baseline achieves relatively good performance at the ratio 10% and then drops to 70.6% Precision@1 and 0.77 MRR at the ratio 20%, and 47.6% Precision@1 and 0.57 MRR at the ratio 30%.

Among all component-scaling experiments, the GUIFetch baseline achieves comparable performance as our W-AE only at the scaling ratio 5%. However, the performance of the GUIFetch baseline drops significantly when the component-scaling ratio increases, and becomes close to 0 at the scaling ratio 20% or higher. This is because of the sensitivity of the GUIFetch's component matching rules (see Section 4.6.2). When the component-scaling ratio is large, the position and size of the corresponding components in the treated UI and the original UI will no longer be close enough under the threshold, and thus will not be matched. For all component-removal experiments, the GUIFetch baseline "unsurprisingly" achieves the perfect performance (all metrics being 1.0). This perfect performance is because all components left in a treated UI are intact and thus can match their counterpart components



(a) Several well-aligned, close-by, same-type components as one component

(b) Similarity of large components overshadows that of small components

(c) Foreground components are overlooked

Figure 4.14: Examples of non-ground-truth (NGT) UI ranked before ground-truth (GT) UI

in the original UI. Furthermore, the GUIFetch's similarity metric considers only the matched versus unmatched components in the query UI. As such, the treated UI and the original UI end up with the similarity score 1.0. However, our generalization study shows that the component-matching heuristics and the similarity metric of GUIFetch do not work well in reality for finding relevant UI designs for real-world query UIs as judged by human.

Retrieval Failure Analysis To gain deeper insight into our CNN model's capability of encoding the visual semantics of UI designs, we manually examine the retrieval-failure cases in which the ground-truth (GT) UI is ranked after other non-ground-truth (NGT) UIs, and identify three main causes for retrieval failures in our automatic experiments. Figure 4.14 shows the typical examples for these three types of retrieval failures.

First, the query UI contains several well-aligned, close-by, same-type components, but the model returns some UIs that have some same-type but bigger and less number of components in the corresponding UI region (Figure 4.14 (a)). This reveals the limitation of our model in distinguishing several well-aligned, close-by, same-type components from one another. However, certain level of modeling fuzziness is important for retrieving similar UI designs with variant numbers of components (such as Figure 4.2 (g) versus (h)/(i)/(j) and the Image Gallery example in Figure 4.16). It is important to note that we use pairs of the original and treated UIs as relevant UI designs in our automatic experiments, and consider all other UIs in the database as "irrelevant" for a query UI. However, as the example in Figure 4.14 (a) shows, the non-ground-truth UIs can still be relevant to the query UIs. Such UI design relevance can only be judged by human, as we do in the generalization experiment and user study.

Second, the query UI contains a UI component (usually an ImageView) covering a large area of the UI, and the model returns some UIs that are similar to the query UI only by that large component, but not similar in other parts of the UI designs (Figure 4.14 (b)). Such retrieval results indicate that our model does not treat the features from small or large visual components equivalently. This inequivalent treatment is reasonable as large components are visually more evident, but it may result in the similarity of large components overshadowing that of small components.

Third, the query UI contains some foreground UI components overlapping a large background component, but the returned UIs contain only the background component without the foreground components, especially when the foreground and background components are of the same type. Overlapping components, especially the same-type ones, can be visually indistinguishable in UI wireframes, because they lack high-fidelity visual effects (e.g., distinct colors or images) to tell them apart. They pose a threat to our wireframe-based UI design search. However, according to our observation, most of UI designs with overlapping components have a background image on top of which real-functional UI components are laid. By removing such background images when generating the UI wireframes, this threat could be mitigated.

In the face of component-scaling and component-removal variations in UI designs, our CNN-based method that models the visual semantics of the whole UI designs significantly outperforms the image-similarity based and the component-matching based methods. But the performance of our method could be further enhanced by the capability of modeling well-aligned, close-by components, small-size components, and overlapping components.

4.7 Generalization Evaluation

To further evaluate our model when applied to real world applications, we conduct a human evaluation of the relevance of the UI design search results. We do this by searching the UI design database using unseen UI designs as queries to confirm the generalization of our model. To this end, we need to construct another dataset of unseen UIs. We introduce this dataset, the human evaluation procedure and the metrics in the following. Based on the performance results of the three baseline methods in our automatic evaluation in Section 4.6.3, we use the GUIFetch baseline in this study. We do not use the color-histogram and SIFT baselines for two reasons. First, our automatic evaluation shows that the color-histogram and SIFT baselines have very poor performance even in face of artificial design variations. Second, human evaluation of the relevance of UI design search results is labor intensive, and considering two more baselines will double the manual evaluation effort.

4.7.1 Dataset of Unseen Query UIs

The UI design database of our tool contains UI screenshots from 25 categories of Android applications. We randomly download one more application per category which have not been included in our proof-of-concept implementation. The same reverse-engineering method [8] is used to obtain the UI screenshots of this newly downloaded application. We generate the corresponding UI wireframes for the collected UI screenshots as described in Section 4.3.1. We select two UI wireframes per application and obtain 50 UI wireframes as the query UI design in this study. The selected UI wireframes contain variant types and numbers of visual components, according to our observation.

4.7.2 Procedure

We recruited five participants, P1, P2, P3, P4 and P5, from our school as human annotators. They have been working on Android app development for at least two years. For each query UI, we obtain the top-10 search results (i.e., 20 UI designs in total) by our method and the GUIFetch baseline respectively. To avoid expectancy bias, these 20 UI designs are randomly mixed together so that the human annotators have no knowledge about which UI design is returned by which method and the ranking of that UI design in the search results. The two annotators examine the UI design search results independently. For each query UI wireframe, they classify each

Table 4.1: Pairwise comparisons of inter-rater agreements

	P1	**P2**	**P3**	**P4**	**P5**
P1	-	0.43	0.37	0.45	0.48
P2	0.43	-	0.38	0.51	0.49
P3	0.37	0.38	-	0.43	0.42
P4	0.45	0.51	0.43	-	0.56
P5	0.48	0.49	0.42	0.56	-

Table 4.2: Results of human relevance evaluation

	Relaxed		**Moderate**		**Strict**	
	W-AE	**GUIFetch**	**W-AE**	**GUIFetch**	**W-AE**	**GUIFetch**
Pre@1	**0.84**	0.64	**0.5**	0.32	0.14	**0.16**
Pre@5	**0.77**	0.65	**0.47**	0.34	**0.20**	0.13
Pre@10	**0.75**	0.62	**0.43**	0.31	**0.15**	0.12
MRR	**0.90**	0.78	**0.62**	0.48	**0.27**	0.24

of the 20 UI designs as relevant or irrelevant to the query UI. The annotators are given the original UI screenshot of the query UI wireframe as a reference for comparison.

4.7.3 Metrics

We use two statistical methods to measure the inter-rater agreement between two human annotators and among all five human annotators. For the first metric, we compute Cohen's kappa statistics [125], which is suitable for measuring the agreement between two raters accessing multiple items into two categories. For the second metric, we compute Fleiss's kappa statistics [126], which is used to evaluate the agreement between multiple raters. Based on the five annotators' judgment of UI design relevance, we regard a returned UI design as relevant by three strategies: strict (all annotators label it as relevant) moderate (the majority of annotators label it as relevant) and relaxed (at least one annotator labels it as relevant). We then compute Precision@k (k=1, 5, 10) and MRR. We do not use Recall and Mean Average Precision (MAP) in this study because it is impossible to manually annotate all relevant UI designs for a query UI in a large UI-design database (54,987 UI screenshots in our proof-of-concept implementation).

4.7.4 Generalization Results

All participants spent about 120 minutes to rate the relevance of the 1000 UI designs to their corresponding query UI wireframes. Table 4.1 shows the Cohen's kappa results of the pairwise comparisons among all five participants. For these comparisons, most of the kappa statistics fall in the range of 0.42-0.56, which indicates a moderate to substantial agreement. We further conducted the Fleiss's kappa [126] to evaluate the agreement among all raters. The Fleiss's kappa for the 2500 (500x5) annotations of UI design search results by our method and the GUIFetch baseline are 0.41 and 0.51, respectively. We consider this level of agreement as acceptable because it can be rather subjective for determining the relevance of UI designs, depending on different background, experience, education and even culture of the human annotators.

According to our observations and interviews, there are four aspects these annotators that are most concerned with, including the semantic meaning of the UI (i.e., functionality), the layout, the types of components and the number of components. Some participants focus more on some aspects while others are more concerned with other parts. Some participants are rather strict while others are relatively relaxed. Different from manual-labelling tasks like image classification, there is no hard right or wrong answer for checking each recommendation result. Therefore, the Cohen's and Fleiss's kappa rates are not so high. However, in summary, the overall feedback quantitative results from all participants still reflect that our method is much better than the baseline as we discuss later.

Table 4.2 shows the performance metrics of our method and the GUIFetch baseline. Our W-AE significantly outperforms the GUIFetch baseline in relaxed and moderate strategies by a large margin, and maintains an advantage over GUIFetch in the strict strategy. By the relaxed strategy, our W-AE has comparable performance as its performance in the scaling-10% and removal-20% experiments (see Figure 4.13). In the moderate strategy, our W-AE still achieves precision@1=0.5 and MRR=0.62. By the strict strategy, our W-AE remains a small advantage over GUIFetch. Since this study involves many variations, it is hard to reach an agreement for five participants. The GUIFetch baseline in reality no longer has the perfect performance as it does in the component-removal experiments. Its performance is also much worse than that of some scaling experiments where the GUIFetch performs well.

Our CNN-based method can robustly retrieve relevant UI designs for a set of diverse, unseen real-application query UIs. In contrast, individual component-matching based heuristics find much fewer relevant UI designs for these real-application query UIs.

4.8 User Study

We then conduct a user study to evaluate the usefulness of our search engine and diversity of the retrieved UI designs. We choose five UI design tasks from Daily UI design challenge[6], recruit 18 students to design these tasks, search relative UIs and modify their draft using our tool, and then rate the usefulness and diversity of the recommendations. In the following, we introduce the details of these five tasks, the experiment procedure and the metrics used in this RQ.

[6] https://www.dailyui.co/

4.8.1 UI Design Tasks

We select five UI design tasks from Daily UI design challenge[6]: *sign-up*, *image gallery*, *login*, *preference setting*, and *navigation drawer*. These five UIs cover essential features of Android mobile applications: *sign-up* and *image gallery* are typical UIs for collecting user inputs and displaying information content, respectively. *login* is a common feature for user authentication, and *preference setting* is commonly used for software customization. *navigation drawer* is a core interaction feature to provide users the access to all app functionalities. Furthermore, these features are easy to understand even for non-professional UI designers who are the targeted users in this study

4.8.2 Procedure

We recruit 18 students from our school through the school's mailing list. Although six students have some front-end software development experience, none of the participants have Android UI design experience. In other word, they are inexperienced designers, same as Lucy in Section 4.2. Participants are given the five UI design tasks and are asked to design a UI wireframe for each task. Each task is allocated 15-30 minutes. Due to the time limitation, we do not ask the participants to design high-fidelity visual effects of the UIs. To assist their design work, the participants use our web tool to draw the UI wireframes and search our database of 54,987 Android UI designs (see Section 4.4). The tool returns the top-10 UI designs for a query UI. We give the participant a tutorial of tool usage and a 15-minute warm-up session to learn to use the tool. For each task, the participants can search as many times as they wish. For the last search, they are asked to select the UI designs in the search results that they consider relevant to the query UI wireframe they draw. They are also asked to rate the overall diversity and usefulness of the search results of the last search by 5-point Likert scale (1 being the lowest and 5 being the highest). In detail, usefulness refers to how useful search results help participants understand/adjust design options if they are facing real UI design tasks. For example, when participants are searching for some UIs related building a sign-up page, the recommendations from our search engine fit into their design requirements. Diversity refers to the diversity of the recommendation results, for example, whether the recommended UIs involve variant component usage/layouts or color/size/font effects which may be beyond their expectations.

4.8.3 Metrics

We record the times of search by the participants for each task. Based on the relevance judgment of UI design search results for the last search, we compute Precision@k (k=1, 5, 10) and MRR. We do not report Recall and MAP as it is impossible to annotate all relevant UI designs in our database of 54,987 UI screenshots for a user-drawn UI wireframe.

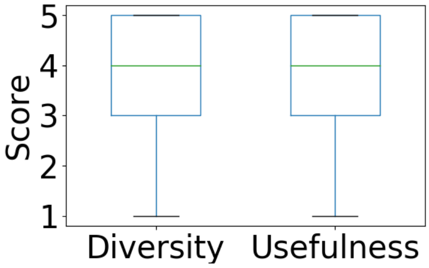


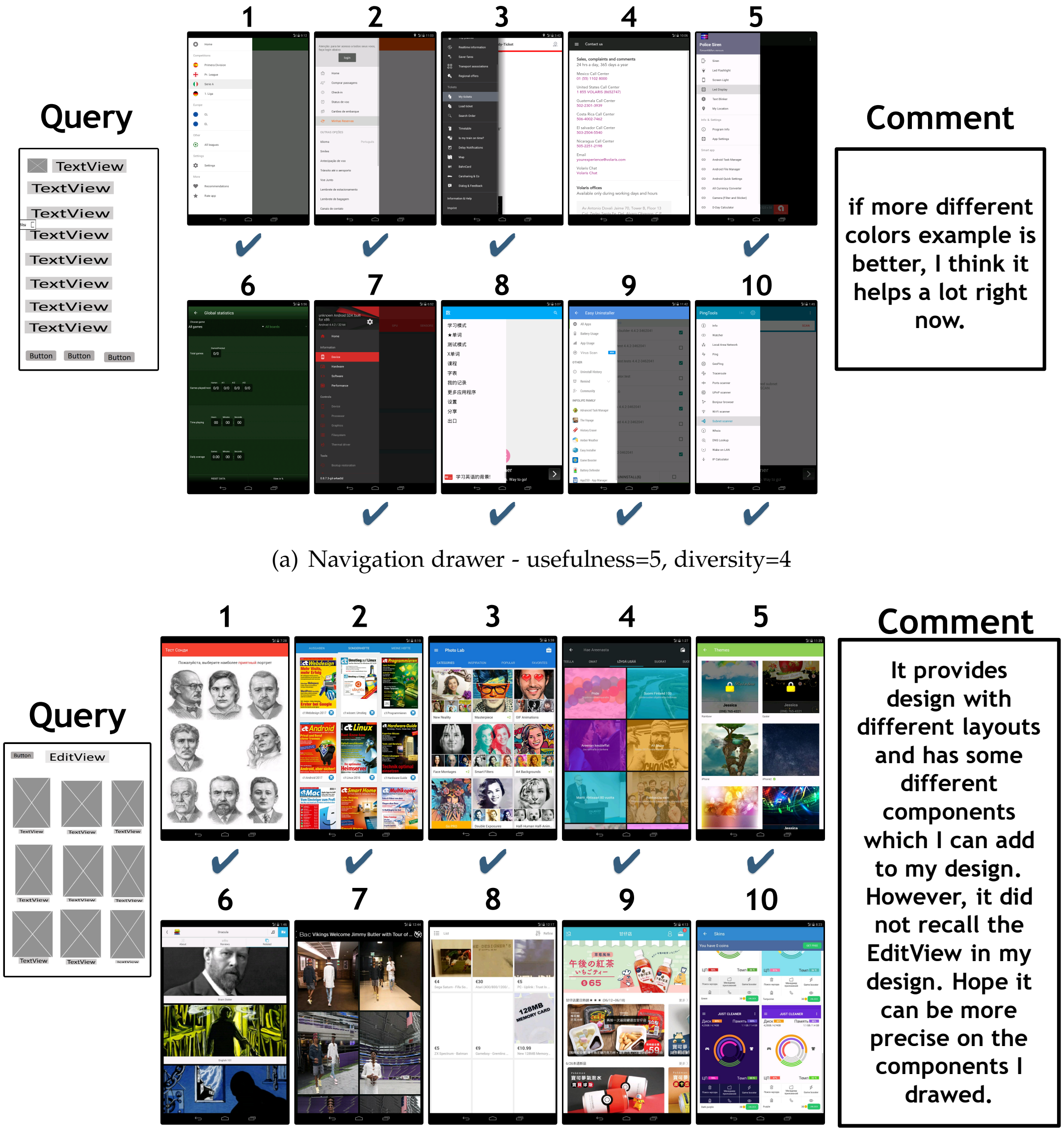
Figure 4.15: Boxplot for diversity and usefulness ratings by participants

4.8.4 Results

The 18 participants perform in total 168 times of search in the five UI design tasks. Among the 90 participant-task sessions, 59 has one search, 12 has two searches, and 20 has three or more searches. The times of search is reasonable considering the short experiment time for each task, as well as the time for drawing UI wireframes. According to the participants' relevance judgment of search results, our W-AE achieves precision@1=0.44, precision@5=0.40, precision@10=0.38, and MRR=0.59. These performance metrics fall in between those for the strict and relaxed strategies in our generation study, which demonstrates the practicality our search engine in support of real-world UI design tasks.

Figure 4.15 shows the boxplot of diversity and usefulness ratings of the search results by the 18 participants. For both these two aspects, the results from our model earn the scores of a median of 4 and the majority of them have a score falling in the range of 3 to 5, which indicates that our UI design search engine is satisfying. Besides,among all 90 searches they rate, the participants rate the search results' diversity at 4 or 5 for 57 (63.3%) searches, and rate the search results' usefulness at 4 or 5 for 51 (56.7%) searches. The motivating scenario illustrated in Section 4.2 is actually derived from the design work by one participant in our user study. We can observe the diversity and usefulness of the search results for inspiring that participant's design of sign-up UI. Figure 4.16 shows two more examples of the search results for the design of navigation drawer and image gallery respectively. For the two user-drawn UI wireframes, our tool returns many relevant UI designs as annotated by the users (highlighted in blue check). Furthermore, the users give 4 or 5 ratings for the diversity and usefulness of the search results, and provide some positive feebacks and useful suggestions on the search results. Even for the irrelevant UI design (e.g., the 4th UI for the query navigation drawer wireframe), our model's recommendation stills makes some sense as that UI is visually similar to the query wireframe. For the image gallery search results, in addition to the top-3 UI designs that have almost the same UI layout as the query UI wireframe, the other returned UI designs demonstrates diverse UI layouts for designing image gallery.

Our search engine does not produce satisfactory search results for 17 searches according to the participants' 1 or 2 diversity and usefulness ratings. By interviewing the participants, we identify two main reasons for unsatisfactory search results.



if more different colors example is better, I think it helps a lot right now.

(a) Navigation drawer - usefulness=5, diversity=4

It provides design with different layouts and has some different components which I can add to my design. However, it did not recall the EditView in my design. Hope it can be more precise on the components I drawed.

(b) Image gallery - usefulness=5, diversity=5

Figure 4.16: Examples of the search results in our user study (check marks indicate that users consider a design relevant)

First, our model tends to return the UI designs that are overall similar to the query UIs. Although this improves the diversity of the search results which is beneficial for gaining design inspirations, it cannot guarantee the presence of some particular UI components or a particular component layout in the returned UI designs that the users want (see the feedback on the search results for image gallery in Figure 4.16). To solve this problem, we may consider more advanced model such as variational autoencoder [127] which can force a greater loss when some user-desired components or component layouts in the query UI do not appear in the search results.

Second, some participants complain that our model are sometimes strict to the location of the components in a UI. For example, when the user draws the switch buttons in the middle region of a preference setting UI, our tool does not return relevant UI designs. But when he moves the switch button to the right side of the UI, our tool can return many relevant preference setting UIs. This example actually shows that our model learns very well the characteristics of preference setting UIs in which switch buttons usually appear on the right side of the UI. Although this modeling capability is desirable to filter out irrelevant UIs, it may make the search of relevant UIs too strict to a particular component layout. To relax the search results, we may use structure similarity of images [128] or attribute graph [129] which support more abstract encoding of the component layout in UIs, and thus more flexible UI design search.

4.9 Threats to Validity

We discuss two types of threats of validity in our work, namely, internal validity and external validity.

4.9.1 Internal Validity

Internal validity refers to the threat that may impact the results to causality [130]. First, our automatic evaluation allows us to conduct large-scale experiments to understand our approach's strengths and weaknesses, but it considers only component-scaling and component-removal variations separately. Real-world UI design variations would be much more complex. However, in order to dive into the influence that each treatment brings, we need to control the variable and it is also not feasible to try every combination of these two treatments. To alleviate this influence, we further conduct generalization experiments and a user study to evaluate our tool by human participants. The performance of our approach in these studies aligns well with that of our automatic evaluation, which gives us confidence in the practicality of our approach for real-world UI design search.

Second, to confirm the generalization of our model, we recruited five students with over two years of experience in Android development to manually examine the results from our model and baselines. However, the notion of the concept of relevance may vary among them and thus impact the results. Some of them may put more emphasis on the semantic meaning of the UI, while some may consider a UI comprised of similar components as relevant. They both make sense since designers may directly reuse the design from the same scenario, but also get inspirations from UIs with similar layout and similar constructions. To keep the evaluation consistent among different participants, we gave them a tutorial to learn the general meaning of these concepts, and a 15-minute warm-up time to get familiar with the tool and the experimental process. Besides, the Cohen's kappa values and the Fleiss's kappa value indicate a moderate agreement between these participants. It is reasonable since the variations we stated above. We involved five students to try and avoid potential bias as best we can and analyse the results in terms of three strategies, namely strict, moderate and relaxed strategies, as states in Section 4.7.3. We assert that by involving five participants and analysing results in terms of these three strategies, this threat to validity is reasonably mitigated. Albeit participants' variance, the overall results still show that our approach outperforms other baselines by aggregating their feedback.

4.9.2 External Validity

External validity refers to the threat that may limit the ability to generalize [130]. First, our data collection tool could collect the majority UI elements from application, but could not capture the detailed HTML elements in the WebView component and some elements in UIs which require some specific engines, such unity3d game engines. Therefore, such limitation may make us lose the UI designs from web components and game UIs. However, the GUI design in these UI which contain HTML elements should be like those UIs which use the native elements. The different implementation is merely an alternative to construct the user interface, while the underlying design principles should be the same. We collect a database of 54,987 UI screenshots from 25 categories of 7,746 top-downloaded Android applications and we believe such large-scale database could cover the majority of UI designs. We let the extension of our tool to collecting UI elements in WebView components and in specific engine as the future work. Second, our approach is general, but our current tool supports only Android UI design search. To further validate the generalizability of our approach, not only should the current tool be further enriched with more UI designs from further Android applications, but it should also be extended to other applications (e.g., iOS, web application). Currently, we have already tested our model/tool on 25 categories of Android apps which demonstrates the generalization of our tool to some extent. As the composition of a user interface is similar in terms of these platforms, we believe that our approach can also be applied with some customization. But this will need to be further explored in the future.

4.10 Conclusion

In this chapter, I present a novel deep learning based approach for UI design search. At the core of our approach is a UI wireframe image autoencoder. Adopting image autoencoder architecture removes the barrier, i.e., labeled relevant UI designs which is impossible to prepare at large scale, for training UI design encoder. Trained using a large database of unlabeled UI wireframes automatically-collected from existing applications, our wireframe encoder learns to encode more abstract and richer visual semantics of the whole UI designs than keywords, low-level image features and component type/position/size matching heuristics, leading to superior performance than the search methods based on these types of primitive information. Our approach demonstrates the promising usefulness in supporting developers to explore and learn about a large UI design space. As the first technique of its kind, our empirical studies also reveal technical and user needs for developing more robust and more usable UI design search methods.

Chapter 5

Predicting Natural Language Labels for Mobile GUI elements

5.1 Introduction

Given millions of mobile apps in Google Play [131] and App store [132], the smart phones are playing increasingly important roles in daily life. They are conveniently used to access a wide variety of services such as reading, shopping, chatting, etc. Unfortunately, many apps remain difficult or impossible to access for people with disabilities. For example, a well-designed user interface (UI) in Figure 5.1 often has elements that don't require an explicit label to indicate their purpose to the user. A checkbox next to an item in a task list application has a fairly obvious purpose for normal users, as does a trash can in a file manager application. However, to users with vision impairment, especially for the blind, other UI cues are needed. According to the World Health Organization(WHO) [20], it is estimated that approximately 1.3 billion people live with some form of vision impairment globally, of whom 36 million are blind. Compared with the normal users, they may be more eager to use the mobile apps to enrich their lives, as they need those apps to represent their eyes. Ensuring full access to the wealth of information and services provided by mobile apps is a matter of social justice [21].

Fortunately, the mobile platforms have begun to support app accessibility by screen readers (e.g., TalkBack in Android [38] and VoiceOver in IOS [39]) for users with vision impairment to interact with apps. Once developers add labels to UI elements in their apps, the UI can be read out loud to the user by those screen readers. For example, when browsing the screen in Figure 5.1 by screen reader, users will hear the clickable options such as "navigate up", "play", "add to queue", etc. for interaction. The screen readers also allow users to explore the view using gestures, while also audibly describing what's on the screen. This is useful for people with vision impairments who cannot see the screen well enough to understand what is there, or select what they need to.

Despite the usefulness of screen readers for accessibility, there is a prerequisite for them functioning well, i.e., the existence of labels for the UI components within the apps. In detail, the Android Accessibility Developer Checklist guides developers

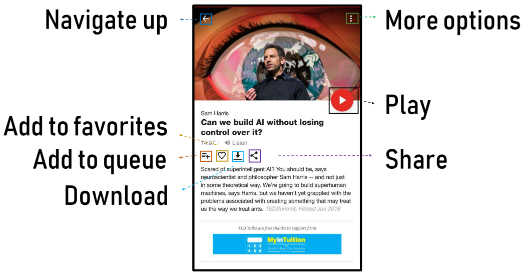
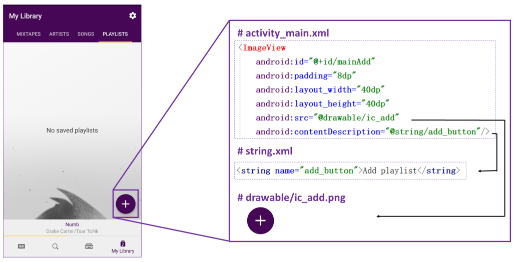


Figure 5.1: Example of UI components and labels.



```
# activity_main.xml
<ImageView
    android:id="@+id/mainAdd"
    android:padding="8dp"
    android:layout_width="40dp"
    android:layout_height="40dp"
    android:src="@drawable/ic_add"
    android:contentDescription="@string/add_button"/>
# string.xml
<string name="add_button">Add playlist</string>

# drawable/ic_add.png
```

Figure 5.2: Source code for setting up labels for "add playlist" button (which is indeed a *clickable ImageView*).

to "*provide content descriptions for UI components that do not have visible text*" [58]. Without such content descriptions[1], the Android TalkBack screen reader cannot provide meaningful feedback to a user to interact with the app.

Although individual apps can improve their accessibility in many ways, the most fundamental principle is adhering to platform accessibility guidelines [58]. However, according to our empirical study in Section 5.2, more than 77% apps out of 10,408 apps miss such labels for the image-based buttons, resulting in the blind's inaccessibility to the apps. Considering that most app designers and developers are of no vision issues, they may lack awareness or knowledge of those guidelines targeting for blind users. To assist developers with spotting those accessibility issues, many practical tools are developed such as Android Lint [22], Accessibility Scanner [23], and other accessibility testing frameworks [24; 25]. However, none of these tools can help fix the label-missing issues. Even if developers or designers can locate these issues, they may still not be aware how to add concise, easy-to-understand descriptions to the GUI components for users with vision impairment. For example, many developers may add label "add" to the button in Figure 5.2 rather than "add playlist" which is more informative about the action.

To overcome those challenges, we develop a deep learning based model to automatically predict the content description. Note that we only target at the image-based buttons in this work as these buttons are important proxies for users to interact with apps, and cannot be read directly by the screen reader without labels. Given the

[1] "labels" and "content description" refer to the same meaning and we use them interchangeably in this paper

UI image-based components, our model can understand its semantics based on the collected big data, and return the possible label to components missing content descriptions. We believe that it can not only assist developers in efficiently filling in the content description of UI components when developing the app, but also enable users with vision impairment to access to mobile apps.

Inspired by image captioning, we adopt the CNN and transformer encoder decoder for predicting the labels based on the large-scale dataset. The experiments show that our LabelDroid can achieve 60.7% exact match and 0.654 ROUGE-L score which outperforms both state-of-the-art baselines. We also demonstrate that the predictions from our model is of higher quality than that from junior Android developers. The experimental results and feedbacks from these developers confirm the effectiveness of our LabelDroid.

Despite the usefulness of screen readers for accessibility, there is a prerequisite for them functioning well, i.e., the existence of labels for the UI components within the apps. In detail, the Android Accessibility Developer Checklist guides developers to "*provide content descriptions for UI components that do not have visible text*" [58]. Without such content descriptions[2], the Android TalkBack screen reader cannot provide meaningful feedback to a user to interact with the app.

Although individual apps can improve their accessibility in many ways, the most fundamental principle is adhering to platform accessibility guidelines [58]. However, according to our empirical study in Section 5.2, more than 77% apps out of 10,408 apps miss such labels for the image-based buttons, resulting in the blind's inaccessibility to the apps. Considering that most app designers and developers are of no vision issues, they may lack awareness or knowledge of those guidelines targeting for blind users. To assist developers with spotting those accessibility issues, many practical tools are developed such as Android Lint [22], Accessibility Scanner [23], and other accessibility testing frameworks [24; 25]. However, none of these tools can help fix the label-missing issues. Even if developers or designers can locate these issues, they may still not be aware how to add concise, easy-to-understand descriptions to the GUI components for users with vision impairment. For example, many developers may add label "add" to the button in Figure 5.2 rather than "add playlist" which is more informative about the action.

To overcome those challenges, we develop a deep learning based model to automatically predict the content description. Note that we only target at the image-based buttons in this work as these buttons are important proxies for users to interact with apps, and cannot be read directly by the screen reader without labels. Given the

UI image-based components, our model can understand its semantics based on the collected big data, and return the possible label to components missing content descriptions. We believe that it can not only assist developers in efficiently filling in the content description of UI components when developing the app, but also enable users with vision impairment to access to mobile apps.

Inspired by image captioning, we adopt the CNN and transformer encoder de-

[2] "labels" and "content description" refer to the same meaning and we use them interchangeably in this paper

coder for predicting the labels based on the large-scale dataset. The experiments show that our LabelDroid can achieve 60.7% exact match and 0.654 ROUGE-L score which outperforms both state-of-the-art baselines. We also demonstrate that the predictions from our model is of higher quality than that from junior Android developers. The experimental results and feedbacks from these developers confirm the effectiveness of our LabelDroid.

Our contributions can be summarized as follow:

- To our best knowledge, this is the first work to automatically predict the label of UI components for supporting app accessibility. We hope this work can invoke the community attention in maintaining the accessibility of mobile apps.
- We carry out a motivational empirical study for investigating how well the current apps support the accessibility for users with vision impairment.
- We construct a large-scale dataset of high-quality content descriptions for existing UI components. We release the dataset[3] for enabling other researchers' future research.

5.2 Motivational Mining Study

While the main focus and contribution of this work is developing a model for predicting content description of image-based buttons, we still carry out an empirical study to understand whether the development team adds labels to the image-based buttons during their app development. The current status of image-based button labeling is also the motivation of this study. But note that this motivational study just aims to provide an initial analysis towards developers supporting users with vision impairment, and a more comprehensive empirical study would be needed to deeply understand it.

5.2.1 Data Collection

To investigate how well the apps support the users with vision impairment, we randomly crawl 19,127 apps from Google Play [131], belonging to 25 categories with the installation number ranging from 1K to 100M.

We adopt the app explorer [8] to automatically explore different screens in the app by various actions (e.g., click, edit, scroll). During the exploration, we take the screenshot of the app GUI, and also dump the run-time front-end code which identifies each element's type (e.g., TextView, Button), coordinates in the screenshots, content description, and other metadata. Note that our explorer can only successfully collect GUIs in 15,087 apps. After removing all duplicates by checking the screenshots, we finally collect 394,489 GUI screenshots from 15,087 apps. Within the collected data, 278,234(70.53%) screenshots from 10,408 apps contain image-based

[3] https://github.com/chenjshnn/LabelDroid

Table 5.1: Statistics of label missing situation

Element	#Miss/#Apps	#Miss/#Screens	#Miss/#Elements
ImageButton	4,843/7,814 (61.98%)	98,427/219,302(44.88%)	241,236/423,172(57.01%)
Clickable Image	5,497/7,421 (74.07%)	92,491/139,831(66.14%)	305,012/397,790(76.68%)
Total	8,054/10,408 (77.38%)	169,149/278,234(60.79%)	546,248/820,962(66.54%)

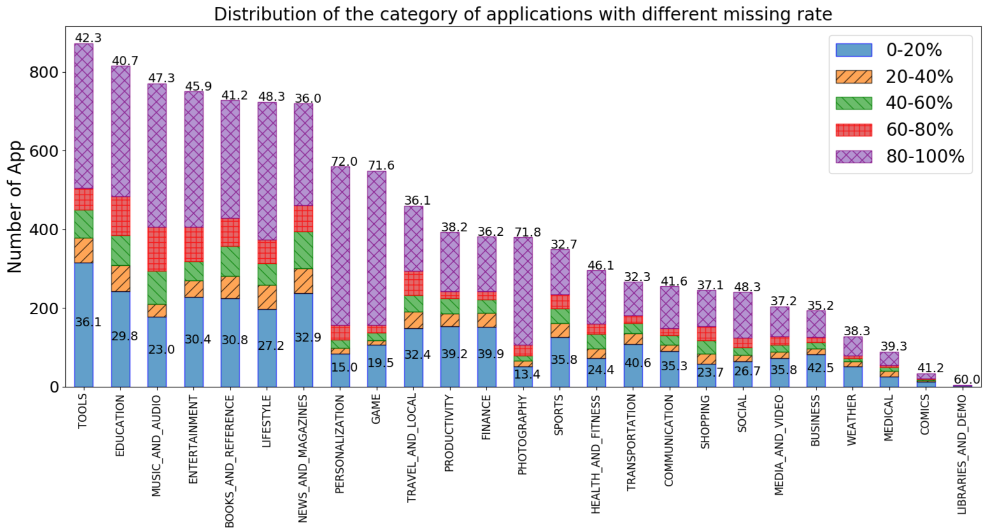


Figure 5.3: The distribution of the category of applications with different rate of image-based buttons missing content description

buttons including clickable images and image buttons, which forms the dataset we analyse in this study.

5.2.2 Current Status of Image-Based Button Labeling in Android Apps

Table 5.1 shows that 4,843 out of 7,814 apps (61.98%) have image buttons without labels, and 5,497 out of 7,421 apps (74.07%) have clickable images without labels. Among all 278,234 screens, 169,149(60.79%) of them including 57.01% image buttons and 76.68% clickable images within these apps have at least one element without explicit labels. It means that more than half of image-based buttons have no labels. These statistics confirm the severity of the button labeling issues which may significantly hinder the app accessibility to users with vision impairment.

We then further analyze the button labeling issues for different categories of mobile apps. As seen in Figure 5.3, the button labeling issues exist widely across different app categories, but some categories have higher percentage of label missing buttons. For example, 72% apps in *Personalization*, 71.6% apps in *Game*, and 71.8% apps in *Photography* have more than 80% image-based button without labels. *Personalization* and *Photography* apps are mostly about updating the screen background, the alignment of app icons, viewing the pictures which are mainly for visual comfort. The *Game* apps always need both the screen viewing and the instantaneous interactions which are rather challenging for visual-impaired users. That is why developers

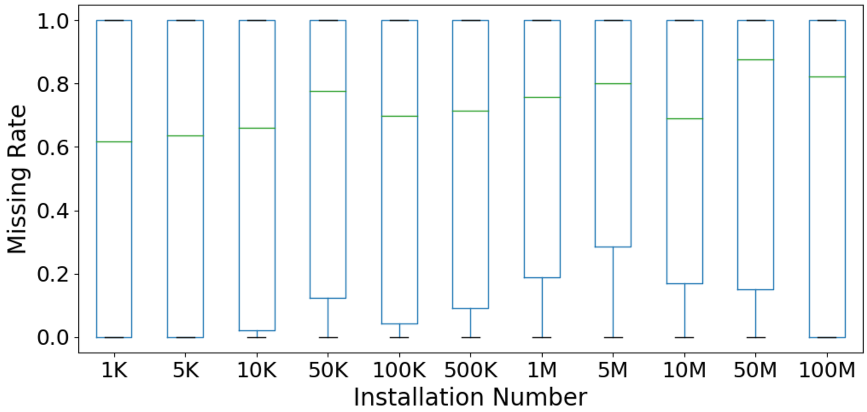


Figure 5.4: Box-plot for missing rate distribution of all apps with different installation numbers.

rarely consider to add labels to image-based buttons within these apps, although these minority users deserve the right for the entertainment. In contrast, about 40% apps in *Finance*, *Business*, *Transportation*, *Productivity* category have relatively more complete labels for image-based buttons with only less than 20% label missing. The reason accounting for that phenomenon may be that the extensive usage of these apps within blind users invokes developers' attention, though further improvement is needed.

To explore if the popular apps have better performance in adding labels to image-based buttons, we draw a box plot of label missing rate for apps with different installation numbers in Figure 5.4. There are 11 different ranges of installation number according to the statistics in Google Play. However, out of our expectation, many buttons in popular apps still lack the labels. Such issues for popular apps may post more negatively influence as those apps have a larger group of audience. We also conduct a Spearman rank-order correlation test [133] between the app installation number and the label-missing rate. The correlation coefficient is 0.046 showing a very weak relationship between these two factors. This result further proves that the accessibility issue is a common problem regardless of the popularity of applications. Therefore, it is worth developing a new method to solve this problem.

Summary: By analyzing 10,408 existing apps crawled from Google Play, we find that more than 77% of them have at least one image-based button missing labels. Such phenomenon is further exacerbated for apps categories highly related to pictures such as personalization, game, photography . However, out of our expectation, the popular apps do not behave better in accessibility than that of unpopular ones. These findings confirm the severity of label missing issues, and motivate our model development for automatic predicting the labels for image-based buttons.

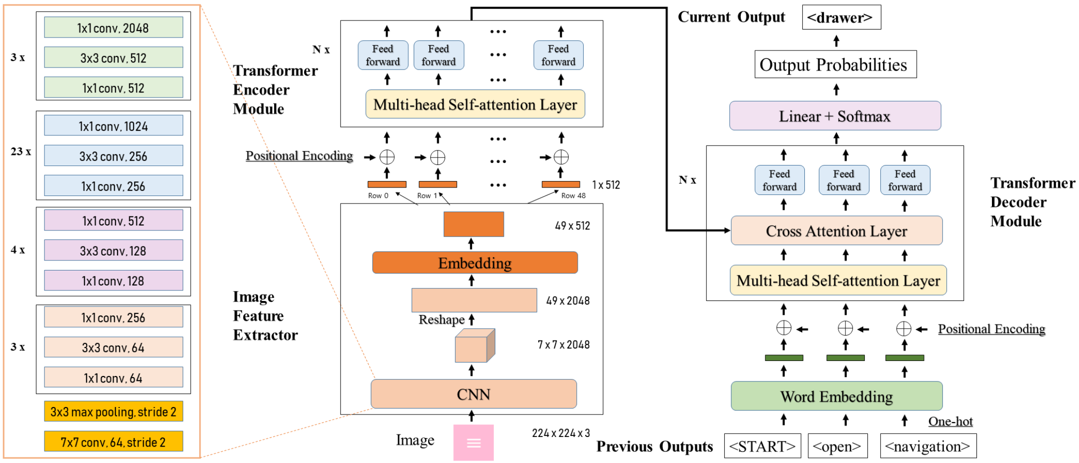


Figure 5.5: Overview of our approach

5.3 Approach

Rendering a UI widget into its corresponding content description is the typical task of image captioning. The general process of our approach is to firstly extract image features using CNN [134], and encode this extracted informative features into a tensor using an encoder module. Based on the encoded information, the decoder module generates outputs (which is a sequence of words) conditioned on this tensor and previous outputs. Different from the traditional image captioning methods based on CNN and RNN model or neural translation based on RNN encoder-decoder [135; 136; 112], we adopt the Transformer model [137] in this work. The overview of our approach can be seen in Figure 5.5.

5.3.1 Visual Feature Extraction

To extract the visual features from the input button image, we adopt the convolutional neural network [138] which is widely used in software engineering domain [94; 118; 139]. CNN-based model can automatically learn latent features from a large image database, which has outperformed hand-crafted features in many computer vision tasks [95; 84]. CNN mainly contains two kinds of layers, i.e., convolutional layers and pooling layers.

Convolutional Layer. A Convolution layer performs a linear transformation of the input vector to extract and compress the salient information of input. Normally, an image is comprised of a $H \times W \times C$ matrix where H, W, C represent height, width, channel of this image respectively. And each value in this matrix is in the range of 0 to 255. A convolution operation uses several small trainable matrices (called kernels) to slide over the image along the width and height in a specified stride. Each movement will compute a value of output at the corresponding position by calculating the sum of element-wise product of current kernel and current sub-matrix of the image. One kernel performs one kind of feature extractor, and computes one layer of the final feature map. For example, for an image $I \in R^{HWC}$, we feed it in a convolutional layer with stride one and k kernels, the output will have the dimension of $H' \times W' \times k$, where H' and W' are the times of movement along height & width.

Pooling Layer. A pooling layer is used to down-sample the current input size to mitigate the computational complexity required to process the input by extracting dominant features, which are invariant to position and rotation, of input. It uses a fixed length of window to slide the image with a fixed stride and summarises current scanned sub-region to one value. Normally, the stride is same as the window's size so that the model could filter meaningless features while maintaining salient ones. There are many kinds of strategy to summarise sub-region. For example, for max-pooling, it takes the maximum value in the sub-region to summarise current patch. For average pooling, it takes the average mean of all values in current patch as the output value.

5.3.2 Visual Semantic Encoding

To further encode the visual features extracted from CNN, we first embed them using a fully-connected layer and then adopt the encoder based on the Transformer model which was first introduced for the machine translation task. We select the Transformer model as our encoder and decoder due to two reasons. First, it overcomes the challenge of long-term dependencies [140], since it concentrates on all relationships between any two input vectors. Second, it supports parallel learning because it is not a sequential learning and all latent vectors could be computed at the same time, resulting in the shorter training time than RNN model. Within the encoder, there are two kinds of sublayers: the multi-head self-attention layer and the position-wise feed forward layer.

Multi-head Self-attention Layer Given a sequence of input vectors $X = [x_1, x_2, ..., x_n]^T (X \in R^{n \times d_{embed}})$, a self-attention layer first computes a set of query (Q), key (K), value (V) vectors ($Q/K \in R^{d_k \times n}, V \in R^{d_v \times n}$) and then calculates the scores for each position vector with vectors at all positions. Note that image-based buttons are artificial images rendered by the compiler with the specified order i.e., from left to right, from top to bottom. Therefore, we also consider the sequential spatial information to capture the dependency between the top-left and bottom-right features extracted by the CNN model. For position i, the scores are computed by taking the dot product of query vector q_i with all key vectors $k_j (j \in 1, 2, ...n)$.

In order to get a more stable gradient, the scores are then divided by $\sqrt{d_k}$. After that, we apply a softmax operation to normalize all scores so that they are added up to 1. The final output of the self-attention layer is to multiply each value vector to its corresponding softmax score and then sums it up. The matrix formula of this procedure is:

$$Self\_Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{5.1}$$

where $Q = W_q^e X^T$, $K = W_k^e X^T$, $V = W_v^e X^T$ and $W_q^e \in R^{d_k \times d_{embed}}$, $W_k^e \in R^{d_k \times d_{embed}}$,

$W_v^e \in R^{d_v \times d_{embed}}$ are the trainable weight metrics. The $softmax(\frac{QK^T}{\sqrt{d_k}})$ can be regarded as how each feature (Q) of the feature sequence from CNN model is influenced by all other features in the sequence (K). And the result is the weight to all features V.

The multi-head self-attention layer uses multiple sets of query, key, value vectors and computes multiple outputs. It allows the model to learn different representation sub-spaces. We then concatenate all the outputs and multiply it with matrix $W_o^e \in R^{d_{model} \times hd_v}$ (where h is the number of heads) to summarise the information of multi-head attention.

Feed Forward Layer Then a position-wise feed forward layer is applied to each position. The feed-forward layers have the same structure of two fully connected layers, but each layer is trained separately. The feed forward layer is represented as:

$$Feed\_forward(Z) = W_2^e \times (W_1^e \times Z + b_1^e) + b_2^e \tag{5.2}$$

where $W_1^e \in R^{d_{ff} \times d_{model}}$, $W_2^e \in R^{d_{model} \times d_{ff}}$, $b_1^e \in R^{d_{ff}}$, $b_2^e \in R^{d_{model}}$ are the trainable parameters of two fully connected layers.

Besides, for each sub-layer, Transformer model applies residual connection [138] and layer normalization [141]. The equation is $LayerNorm(x + Sublayer(x))$, where x and Sublayer(x) are the input and output of current sub-layer. For input embedding, Transformer model also applies position encoding to encode the relative position of the sequence.

5.3.3 Content Description Generation

As mentioned in Section 5.3.2, the encoder module is comprised of N stacks of layers and each layer consists of a multi-head self-attention sub-layer and a feed-forward layer. Similar to the encoder, the decoder module is also of M-stack layers but with two main differences. First, an additional cross-attention layer is inserted into the two sub-layers. It takes the output of top-layer of the encoder module to compute query and key vectors so that it can help capture the relationship between inputs and targets. Second, it is masked to the right in order to prevent attending future positions.

Cross-attention Layer. Given current time t, max length of content description L, previous outputs $S^d (\in R^{d_{model} \times L})$ from self-attention sub-layer and output $Z^e (\in R^{d_{model} \times n})$ from encoder module, the cross-attention sub-layer can be formulated as:

$$Cross\_Attention(Q^e, K^e, V^d) = softmax(\frac{Q^e (K^e)^T}{\sqrt{d_k}})V^d \tag{5.3}$$

where $Q^e = W_q^d Z$, $K^e = W_k^d Z$, $V^d = W_v^d S^d$, $W_q^d \in R^{d_k \times d_{model}}$, $W_k^d \in R^{d_k \times d_{model}}$ and $W_v^d \in R^{d_v \times d_{model}}$. Note that we mask $S_k^d = 0$ (for $k \geq t$) since we currently do not know future values.

Final Projection. After that, we apply a linear softmax operation to the output of the top-layer of the decoder module to predict next word. Given output $D \in R^{d_{model} \times L}$ from decoder module, we have $Y' = Softmax(D * W_o^d + b_o^d)$ and take the t_{th} output

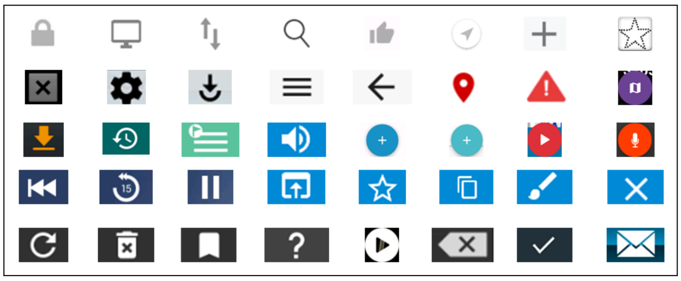


Figure 5.6: Example of our dataset

of Y′ as the next predicted word. During training, all words can be computed at the same time by masking $S_k^d = 0$ (for $k \geq t$) with different t. Note that while training, we compute the prediction based on the ground truth labels, i.e, for time t, the predicted word y'_t is based on the ground truth sub-sequence $[y_0, y_1, ..., y_{t-1}]$. In comparison, in the period of inference (validation/test), we compute words one by one, based on previous predicted words, i.e., for time t, the predicted word y'_t is based on the ground truth sub-sequence $[y'_0, y'_1, ..., y'_{t-1}]$.

Loss Function. We adopt Kullback-Leibler (KL) divergence loss [142] (also called as relative entropy loss) to train our model. It is a natural method to measure the difference between the generated probability distribution q and the reference probability distribution p. Note that there is no difference between cross entropy loss and KL divergence since $D_{kl}(p|q) = H(p,q) - H(p)$, where H(p) is constant.

5.4 Implementation

To implement our model, we further extract the data analysed in Section 5.2 by filtering out the noisy data for constructing a large-scale pairs of image-based buttons and corresponding content descriptions for training and testing the model. Then, we introduce the detailed parameters and training process of our model.

5.4.1 Data Preprocessing

For each screenshot collected by our tool, we also dump the runtime XML which includes the position and type of all GUI components within the screenshot. To obtain all image-based buttons, we crop the GUI screenshot by parsing the coordinates in the XML file. However, the same GUI may be visited many times, and different GUIs within one app may share the same components. For example, a menu icon may appear in the top of all GUI screenshots within the app. To remove duplicates, we first remove all repeated GUI screenshots by checking if their corresponding XML files are the same. After that, we further remove duplicate image-based buttons if they are exactly same by the pixel value. But duplicate buttons may not be 100% same in pixels, so we further remove duplicate buttons if their coordinate bounds and labels are the same because some buttons would appear in a fixed position but

Table 5.2: Details of our accessibility dataset.

	#App	#Screenshot	#Element
Train	6,175	10,566	15,595
Validation	714	1,204	1,759
Test	705	1,375	1,879
Total	7,594	13,145	19,233

with a different background within the same app. For example, for the "back" button at the top-left position in Figure 5.1, once user navigates to another news page, the background of this button will change while the functionality remains.

Apart from the duplicate image-based buttons, we also remove low-quality labels to ensure the quality of our training model. For all content description, we first transform them into lower case and remove punctuation. Then, for non-english languages, we translate them using google translation API. We manually observe 500 randomly selected image-based buttons from Section 5.2, and summarise three types of meaningless labels. First, the labels of some image-based buttons contain the class of elements such as "image button", "button with image", etc. Second, the labels contain the app's name. For example, the label of all buttons in the app Ringtone Maker is "ringtone maker". Third, some labels may be some unfinished placeholders such as "test", "content description", "untitled", "none". We write the rules to filter out all of them, and the full list of meaningless labels can be found in our website.

After removing the non-informative labels, we translate all non-English labels of image-based buttons to English by adopting the Google Translate API. For each label, we add $< start >$, $< end >$ tokens to the start and the end of the sequence. We also replace the low-frequency words with an $< unk >$ token. To enable the mini-batch training, we need to add a $< pad >$ token to pad the word sequence of labels into a fixed length. Note that the maximum number of words for one label is 15 in this work.

After the data cleaning, we finally collect totally 19,233 pairs of image-based buttons and content descriptions from 7,594 apps. Note that the app number is smaller than that in Section 5.2 as the apps with no or uninformative labels are removed. We split cleaned dataset into training, validation[4] and testing set. For each app category, we randomly select 80% apps for training, 10% for validation and the rest 10% for testing. Table 5.2 shows that, there are 15,595 image-based buttons from 6,175 apps as the training set, 1,759 buttons from 714 apps as validation set and 1,879 buttons from 705 apps as testing set. The dataset can also be downloaded from our site.

5.4.2 Model Implementation

We use ResNet-101 architecture [138] pretrained on ImageNet dataset [143] as our CNN module. As you can see in the leftmost of Figure 5.5, it consists of a convolution

[4] for tuning the hyperparameters and preventing the overfitting

layer, a max pooling layer, four types of blocks with different numbers of block (denoted in different colors). Each type of block is comprised of three convolutional layers with different settings and implements an identity shortcut connection which is the core idea of ResNet. Instead of approximating the target output of current block, it approximates the residual between current input and target output, and then the target output can be computed by adding the predicted residual and the original input vector. This technique not only simplifies the training task, but also reduces the number of filters. In our model, we remove the last global average pooling layer of ResNet-101 to compute a sequence of input for the consequent encoder-decoder model.

For transformer encoder-decoder, we take $N = 3$, $d_{embed} = d_k = d_v = d_{model} = 512$, $d_{ff} = 2048$, $h = 8$. We train the CNN and the encoder-decoder model in an end-to-end manner using KL divergence loss [142]. We use Adam optimizer [144] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$ and change the learning rate according to the formula $learning\_rate = d_{model}^{-0.5} \times min(step\_num^{-0.5}, step\_num \times warmup\_steps^{-1.5})$ to train the model, where $step\_num$ is the current iteration number of training batch and the first $warm\_up$ training step is used to accelerate training process by increasing the learning rate at the early stage of training. Our implementation uses PyTorch [145] on a machine with Intel i7-7800X CPU, 64G RAM and NVIDIA GeForce GTX 1080 Ti GPU.

5.5 Accuracy Evaluation

We first evaluate the accuracy of our proposed tool, LabelDroid, with automated testing. We use randomly selected 10% apps including 1,879 image-based buttons as the test data for accuracy evaluation. None of the test data appears in the model training.

5.5.1 Evaluation Metric

To evaluate the performance of our model, we adopt five widely-used evaluation metrics including exact match, BLEU [146], METEOR [147], ROUGE [148], CIDEr [149] inspired by related works about image captioning. The first metric we use is *exact match rate*, i.e., the percentage of testing pairs whose predicted content description exactly matches the ground truth. Exact match is a binary metric, i.e., 0 if any difference, otherwise 1. It cannot tell the extent to which a generated content description differs from the ground-truth. For example, the ground truth content description may contain 4 words, but no matter one or 4 differences between the prediction and ground truth, exact match will regard them as 0. Therefore, we also adopt other metrics. BLEU is an automatic evaluation metric widely used in machine translation studies. It calculates the similarity of machine-generated translations and human-created reference translations (i.e., ground truth). BLEU is defined as the product of n-gram precision and brevity penalty. As most content descriptions for image-based buttons are short, we measure BLEU value by setting n as 1, 2, 3, 4, represented as BLEU@1, BLEU@2, BLEU@3, BLEU@4.

METEOR [147] (Metric for Evaluation of Translation with Explicit ORdering) is another metric used for machine translation evaluation. It is proposed to fix some disadvantages of BLEU which ignores the existence of synonyms and recall ratio. ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [148] is a set of metric based on recall rate, and we use ROUGE-L, which calculates the similarity between predicted sentence and reference based on the longest common subsequence (short for LCS). CIDEr (Consensus-Based Image Description Evaluation) [149] uses term frequency inverse document frequency (tf-idf) [150] to calculate the weights in reference sentence s_{ij} for different n-gram w_k because it is intuitive to believe that a rare n-grams would contain more information than a common one. We use CIDEr-D, which additionally implements a length-based gaussian penalty and is more robust to gaming. We then divide CIDEr-D by 10 to normalize the score into the range between 0 and 1. We still refer CIDEr-D/10 to CIDEr for brevity.

All of these metrics give a real value with range [0,1] and are usually expressed as a percentage. The higher the metric score, the more similar the machine-generated content description is to the ground truth. If the predicted results exactly match the ground truth, the score of these metrics is 1 (100%). We compute these metrics using coco-caption code [151].

5.5.2 Baselines

We set up two state-of-the-art methods which are widely used for image captioning as the baselines to compare with our content description generation method. The first baseline is to adopt the CNN model to encode the visual features as the encoder and adopt a LSTM (long-short term memory unit) as the decoder for generating the content description [140; 152]. The second baseline also adopt the encoder-decoder framework. Although it adopts the CNN model as the encoder, but uses another CNN model for generating the output [153] as the decoder. The output projection layer in the last CNN decoder performs a linear transformation and softmax, mapping the output vector to the dimension of vocabulary size and getting word probabilities. Both methods take the same CNN encoder as ours, and also the same datasets for training, validation and testing. We denote two baselines as *CNN+LSTM*, *CNN+CNN* for brevity.

5.5.3 Results

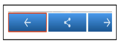
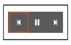
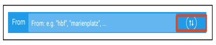
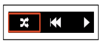
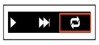
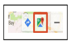
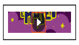
5.5.3.1 Overall Performance

Table 5.3 shows the overall performance of all methods. The performance of two baselines are very similar and they achieve 58.4% and 57.4% exactly match rate respectively. But CNN+LSTM model is slightly higher in other metrics. In contrast with baselines, the generated labels from our LabelDroid for 60.7% image-based buttons exactly match the ground truth. And the average BLEU@1, BLEU@2,

Table 5.3: Results of accuracy evaluation

Method	Exact match	BLEU@1	BLEU@2	BLEU@3	BLEU@4	METEOR	ROUGE-L	CIDEr
CNN+LSTM	58.4%	0.621	0.600	0.498	0.434	0.380	0.624	0.287
CNN+CNN	57.4%	0.618	0.596	0.506	0.473	0.374	0.617	0.284
LabelDroid	**60.7%**	**0.651**	**0.626**	**0.523**	**0.464**	**0.399**	**0.654**	**0.302**

Table 5.4: Examples of wrong predictions in baselines

ID	E1	E2	E3	E4	E5	E6	E7
Button							
CNN+LSTM	start	play	*< unk >*	cycle shuffle mode	cycle repeat mode	color swatch	<unk>
CNN+CNN	next	previous track	call	cycle repeat mode	next	open drawer	<unk> trip check
LabelDroid	back	previous track	exchange origin and destination points	cycle shuffle mode	cycle repeat mode	open in google maps	watch

BLEU@3, BLEU@4, METEOR, ROUGE-L and CIDEr of our method are 0.651, 0.626, 0.523, 0.464, 0.399, 0.654, 0.302. Compared with the two state-of-the-art baselines, our LabelDroid outperforms in all metrics and gains about 2% to 11.3% increase. We conduct the Mann–Whitney U test [154] between these three models among all testing metrics. Since we have three inferential statistical tests, we apply the Benjamini & Hochberg (BH) method [155] to correct p-values. Results show the improvement of our model is significant in all comparisons (p-value<0.01)[5].

To show the generalization of our model, we also calculate the performance of our model in different app categories as seen in Figure 5.7. We find that our model is not sensitive to the app category i.e., steady performance across different app categories. In addition, Figure 5.7 also demonstrates the generalization of our model. Even if there are very few apps in some categories (e.g., medical, personalization, libraries and demo) for training, the performance of our model is not significantly degraded in these categories.

5.5.3.2 Qualitative Performance with Baselines

To further explore why our model behaves better than other baselines, we analyze the image-buttons which are correctly predicted by our model, but wrongly predicted by baselines. Some representative examples can be seen in Table 5.4 as the qualitative observation of all methods' performance. In general, our method shows the capacity to generate different lengths of labels while CNN+LSTM prefers medium length labels and CNN+CNN tends to generate short labels.

Our model captures the fine-grained information inside the given image-based button. For example, the CNN+CNN model wrongly predict "next" for the "back" button (E1), as the difference between "back" and "next" buttons is the arrow direction. It also predicts "next" for the "cycle repeat model" button (E5), as there is one right-direction arrow. Similar reasons also lead to mistakes in E2.

[5] The detailed p-values are listed in https://github.com/chenjshnn/LabelDroid

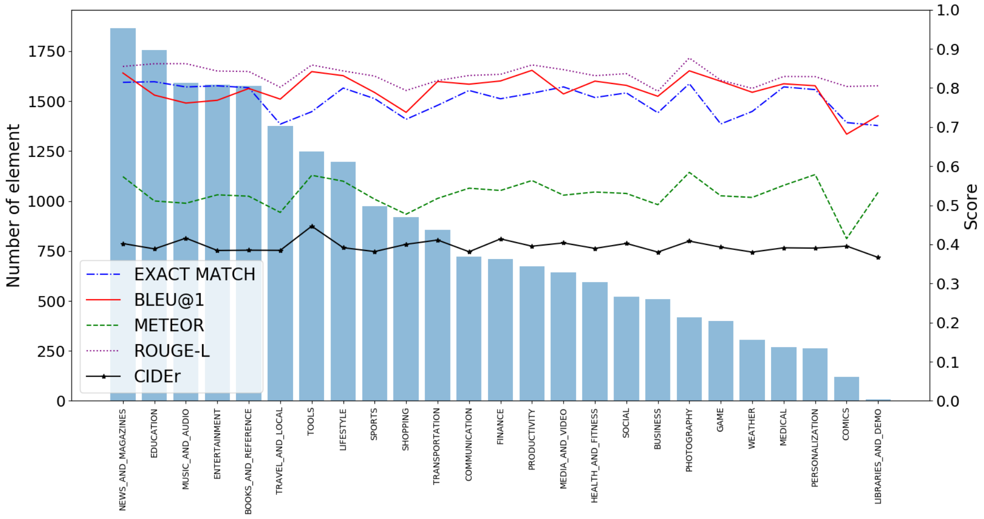


Figure 5.7: Performance distribution of different app category

Our model is good at generating long-sequence labels due to the self-attention and cross-attention mechanisms. Such mechanism can find the relationship between the patch of input image and the token of output label. For example, our model can predict the correct labels such as “exchange origin and destination points” (E3), “open in google maps” (E6). Although CNN+LSTM can generate correct labels for E4, E5, it does not work well for E3 and E6.

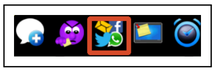
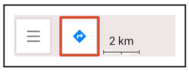
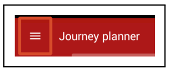
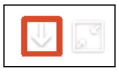
In addition, our model is more robust to the noise than the baselines. For example, although there is “noisy” background in E7, our model can still successfully recognize the “watch” label for it. In contrast, the CNN+LSTM and CNN+CNN are distracted by the colorful background information with “<unk>” as the output.

5.5.3.3 Common Causes for Generation Errors

We randomly sample 5% of the wrongly generated labels for the image-based buttons. We manually study the differences between these generated labels and their ground truth. Our qualitative analysis identifies three common causes of the generation errors.

(1) Our model makes mistakes due to the characteristics of the input. Some image-based buttons are very special and it is totally different from the training data. For example, the E1 in Table 5.5 is rather different from the normal social-media sharing button. It aggregates the icons of whatsapp, twitter, facebook and message with some rotation and overlap. Some buttons are visually similar to others but with totally different labels. The “route” button (E2) in Table 5.5 includes a right arrow which also frequently appears in “next” button. (2) Our prediction is an alternative to the ground truth for certain image-based buttons. For example, the label of E3 is the “menu”, but our model predicts “open navigation drawer”. Although the prediction is totally different from the ground truth in term of the words, they

Table 5.5: Common causes for generation failure.

ID	E1	E2	E3	E4
Cause	Special case	Model error	Alternative	Wrong ground truth
Button				
LabelDroid	*< unk >*	next	open navigation drawer	download
Ground truth	share note	route	menu	story content image

convey the same meaning which can be understood by the blind users. (3) A small amount of ground truth is not the right ground truth. For example, some developers annotate the E4 as "story content image" although it is a "download" button. Although our prediction is different from the ground truth, we believe that our generated label is more suitable for it. This observation also indicates the potential of our model in identifying wrong/uninformative content description for image-based buttons. We manually allocate the 5% (98) failure cases of our model into these three reasons. 54 cases account for model errors especially for special cases, 41 cases are alternatives labels to ground truth, and the last three cases are right but with a wrong groundtruth. It shows that the accuracy of our model is highly underestimated.

5.6 Generalization and Usefulness Evaluation

To further confirm the generalization and usefulness of our model, we randomly select 12 apps in which there are missing labels of image-based buttons. Therefore, all data of these apps do not appear in our training/testing data. We would like to see the labeling quality from both our LabelDroid and human developers.

5.6.1 Procedures

To ensure the representativeness of test data, 12 apps that we select have at least 1M installations (popular apps often influence more users), with at least 15 screenshots. These 12 apps belong to 10 categories. We then crop elements from UI screenshots and filter out duplicates by comparing the raw pixels with all previous cropped elements. Finally, we collect 156 missing-label image-based buttons, i.e., 13 buttons in average for each app.

All buttons are fed to our model for predicting their labels (denoted as M). To compare the quality of labels from our LabelDroid and human annotators, we recruit three PhD students and research staffs (denoted as A1, A2, A3) from our school to create the content descriptions for all image-based buttons. All of them have at least one-year experience in Android app development, so they can be regarded as junior app developers. Before the experiment, they are required to read the accessibility guidelines [58; 156] and we demo them the example labels for some randomly selected image-based buttons (not in our dataset). During the experiment, they are

Table 5.6: The acceptability score (AS) and the standard deviation for 12 completely unseen apps. * denotes $p < 0.05$.

ID	**Package name**	**Category**	**#Installation**	**#Image-based button**	**AS-M**	**AS-A1**	**AS-A2**	**AS-A3**
1	com.handmark.sportcaster	sports	5M - 10M	8	4.63(0.48)	3.13(0.78)	3.75(1.20)	4.38(0.99)
2	com.hola.launcher	personalization	100M - 500M	10	4.40(0.92)	3.20(1.08)	3.50(1.75)	3.40(1.56)
3	com.realbyteapps.moneymanagerfree	finance	1M - 5M	24	4.29(1.10)	3.42(1.29)	3.75(1.45)	3.83(1.55)
4	com.jiubang.browser	communication	5M - 10M	11	4.18(1.34)	3.27(1.21)	3.73(1.54)	3.91(1.38)
5	audio.mp3.music.player	media_and_video	5M - 10M	26	4.08(1.24)	2.85(1.06)	2.81(1.62)	3.50(1.62)
6	audio.mp3.mp3player	music_and_audio	1M - 5M	16	4.00(1.27)	2.75(1.15)	3.31(1.53)	3.25(1.39)
7	com.locon.housing	lifestyle	1M - 5M	10	4.00(0.77)	3.50(1.12)	3.60(1.28)	4.40(0.80)
8	com.gau.go.launcherex.gowidget.weatherwidget	weather	50M - 100M	12	3.42(1.66)	2.92(1.38)	3.00(1.78)	3.42(1.80)
9	com.appxy.tinyscanner	business	1M - 5M	13	3.85(1.23)	3.31(1.20)	3.08(1.59)	3.38(1.44)
10	com.jobkorea.app	business	1M - 5M	15	3.60(1.67)	3.27(1.57)	3.13(1.67)	3.60(1.54)
11	browser4g.fast.internetwebexplorer	communication	1M - 5M	4	3.25(1.79)	2.00(0.71)	2.50(1.12)	2.50(1.66)
12	com.rcplus	social	1M - 5M	7	3.14(1.55)	2.00(1.20)	2.71(1.58)	3.57(1.29)
	AVERAGE			13	3.97*(1.33)	3.06(1.26)	3.27(1.60)	3.62(1.52)

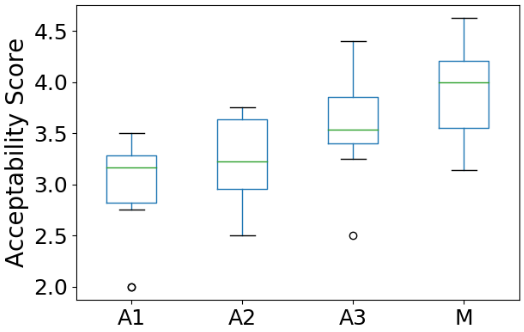
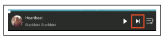
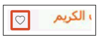
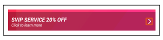
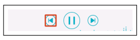
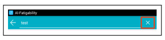


Figure 5.8: Distribution of app acceptability scores by human annotators (A1, A2, A3) and the model (M).

shown the target image-based buttons highlighted in the whole UI (similar to Figure 5.1), and also the meta data about the app including the app name, app category, etc. All participants carried out experiments independently without any discussions with each other.

As there is no ground truth for these buttons, we recruit one professional developer (evaluator) with prior experience in accessibility service during app development to manually check how good are the annotators' comments. Instead of telling if the result is right or not, we specify a new evaluation metric for human evaluators called acceptability score according to the acceptability criterion [157]. Given one predicted content description for the button, the human evaluator will assign 5-point Likert scale [158; 159]) with 1 being least satisfied and 5 being most satisfied. Each result from LabelDroid and human annotators will be evaluated by the evaluator, and the final acceptability score for each app is the average score of all its image-based buttons. Note that we do not tell the human evaluator which label is from developers or our model to avoid potential bias. To guarantee if the human evaluators are capable and careful during the evaluation, we manually insert 4 cases which contain 2 intentional wrong labels and 2 suitable content description (not in 156 testing set) which are carefully created by all authors together. After the experiment, we ask them to give some informal comments about the experiment, and we also briefly introduce LabelDroid to developers and the evaluator and get some feedback from them.

Table 5.7: Examples of generalization.

ID	E1	E2	E3	E4	E5
Button					
M	next song	add to favorites	open ad	previous song	clear query
A1	change to the next song in playlist	add the mp3 as favorite	show more details about SVIP	paly the former one	clean content
A2	play the next song	like	check	play the last song	close
A3	next	like	enter	last	close

5.6.2 Results

Table 5.6[6] summarizes the information of the selected 12 apps and the acceptability scores of the generated labels. The average acceptability scores for three developers vary much from 3.06 (A1) to 3.62 (A3). But our model achieves 3.97 acceptability score which significantly outperforms three developers by 30.0%, 21.6%, 9.7%. The evaluator rates 51.3% of labels generated from LabelDroid as highly acceptable (5 point), as opposed to 18.59%, 33.33%, 44.23% from three developers. Figure 5.8 shows that our model behaves betters in most apps compared with three human annotators. These results show that the quality of content description from our model is higher than that from junior Android app developers. Note that the evaluator is reliable as both 2 intentional inserted wrong labels and 2 good labels get 1 and 5 acceptability score as expected.

To understand the significance of the differences between four kinds of content description, we carry out the Wilcoxon signed-rank test [160] between the scores of our model and each annotator and between the scores of any two annotators. It is the non-parametric version of the paired T-test and widely used to evaluate the difference between two related paired sample from the same probability distribution. The test results suggest that the generated labels from our model are significantly better than that of developers (p-value < 0.01 for A1, A2, and < 0.05 for A3)[7].

For some buttons, the evaluator gives very low acceptability score to the labels from developers. According to our observation, we summarise four reasons accounting for those bad cases and give some examples in Table 5.7. (1) Some developers are prone to write long labels for image-based buttons like the developer A1. Although the long label can fully describe the button (E1, E2), it is too verbose for blind users especially when there are many image-based buttons within one page. (2) Some developers give too short labels which may not be informative enough for users. For example, A2 and A3 annotate the "add to favorite" button as "like" (E2). Since this button will trigger an additional action (add this song to favorite list), "like" could not express this meaning. The same reason applies to A2/A3's labels for E2 and such short labels do not contain enough information. (3) Some manual labels may be ambiguous which may confuse users. For example, A2 and A3 annotate "play the last song" or "last" to "previous song" button (E4) which may mislead users that clicking this button will come to the final song in the playlist. (4) Developers may make

[6] Detailed results are at https://github.com/chenjshnn/LabelDroid

[7] The p-values are adjusted by Benjamin & Hochberg method [155]. All detailed p-values are listed in https://github.com/chenjshnn/LabelDroid

mistakes especially when they are adding content descriptions to many buttons. For example, A2/A3 use "close" to label a "clear query" buttons (E5). We further manually check 135 low-quality (acceptability score = 1) labels from annotators into these four categories. 18 cases are verbose labels, 21 of them are uninformative, six cases are ambiguous which would confuse users, and the majority, 90 cases are wrong.

We also receive some informal feedback from the developers and the evaluator. Some developers mention that one image-based button may have different labels in different context, but they are not very sure if the created labels from them are suitable or not. Most of them never consider adding the labels to UI components during their app development and curious how the screen reader works for the app. All of them are interested in our LabelDroid and tell that the automatic generation of content descriptions for icons will definitely improve the user experience in using the screen reader. All of these feedbacks indicate their unawareness of app accessibility and also confirm the value of our tool.

5.7 Threats to Validity

We discuss two types of threats, i.e., internal validity and external validity.

5.7.1 Internal Validity

Threats to internal validity indicate the potential casual factors that may affect the results. First, our automatic empirical study enables us to understand the current accessibility issues in Android platform. However, the collected apps may be biased and the collected UIs may not be comprehensive due to the limited time of exploration and failure to access certain UIs. To mitigate this potential bias, we collected a large-scale dataset comprising of 391,489 UIs from 10,408 apps among 25 app categories. We tried to collect as many apps as possible to mitigate the bias. We also evaluated the results in different dimensions qualitatively and quantitatively to spot and understand any anomalies. By checking the statistics and the exact data, we found our results can be well-explained and reasonable, which gives us the confidence to report them.

Second, to evaluate the generalisability and usefulness of our proposed techniques, we randomly selected 12 apps as our experimental data, recruited three students to create labels and one to evaluate the created/generated labels from participants and our LabelDroid. The selection of the target apps may lead to biased conclusion as this dataset is small-scale. To alleviate this, we sampled apps of diverse app categories and only considered apps with at least 1M installations. The quantitative and qualitative results also align well with the results in our accuracy evaluation, which demonstrates the soundness of the conclusion. In addition, while we only recruited one evaluator, their view may introduce some personal biases. To mitigate this, the evaluator we picked has prior experience in accessibility service. We also deliberately inserted some wrong and good labels into the labels to check the quality of the evaluator. They passed all the tests. Moreover, we additionally examined the reasons that they rated higher or lower to see if there are any mis-ratings. All ratings are reasonable and truth-worthy based on our observation, which mitigate the potential bias brought by only one evaluator.

5.7.2 External Validity

Threats to external validity focus on the generalisation of the results. We conducted our empirical study in a large-scale UI dataset on the Android platform. While our approach is generic, the dataset may limit the ability to generalize to other platforms, such as desktop apps and websites. However, existing websites and desktop apps also tend to use the simplified icons (i.e., image-based buttons) to provide a minimalist design style, and the website versions and mobile app versions of the same service/app share a same set of icons. For example, people can easily find that YouTube has thumb-up and thumb-down icons that are the same as icons in their mobile apps. While we do not conduct experience on this kind of dataset, we believe our tool can be generalized to these simplified icons. However, we also note that as desktop apps and websites have a larger space compared to a small smartphone, designers may use complicated or compound icons to demonstrate a complicated meaning. For example, the designer may adopt a complicated camera icon with many fine details to mimic the real texture of a camera. Our model may fail in such cases as these kinds of complicated icons may have a different distribution from the icons in our dataset. More experiments can be conducted to evaluate the generalisability of our proposed techniques. In comparison, we believe that while we only experiment on the Android platform, our model should be easily applied to iOS mobile apps as these two platforms show many common icon usages. But this will need to be further explored and confirmed. We release all our code, trained models and dataset to assist in following studies on this.

5.8 Conclusion and Future Work

Engaging the minority especially the disability into the world is a kind of social good. Although millions of apps in the smart phone provide such a chance, the serious accessibility issues in mobile apps hinders the usage. In this chapter, we find that more than 77% apps have at least one image-based button without natural-language label which can be read for users with vision impairment. Considering that most app designers and developers are of no vision issues, they may not understand how to write suitable labels. To overcome this problem, we propose a deep learning model based on CNN and transformer encoder-decoder for learning to predict the label of given image-based buttons. The evaluation demonstrates its accuracy compared with baselines, and also generalization and usefulness by the user study.

We hope that this work can invoke community attention in app accessibility. In the future, we will first improve our model for achieving better quality by taking the app metadata into the consideration. Second, we will also try to test the quality of existing labels by checking if the description is concise and informative.

Chapter 6

Object Detection for Graphical User Interface

6.1 Introduction

While Chapter 4 and Chapter 5 propose two methods to help the UI design prototyping and UI design implementation process, they have a key limitation, i.e., these two methods assume that we can obtain the view hierarchy information of UI designs. In this chapter, I will first discuss the limitation of the assumption and propose a fundamental technique that can fill this gap.

GUI allows users to interact with software applications through graphical elements such as widgets, images and text. Recognizing GUI elements in a GUI is the foundation of many software engineering tasks, such as GUI automation and testing [26; 27; 28; 29], supporting advanced GUI interactions [30; 31], GUI search [32; 17], and code generation [33; 34; 8]. Recognizing GUI elements can be achieved by instrumentation-based or pixel-based methods. Instrumentation-based methods [161; 162; 163] are intrusive and requires the support of accessibility APIs [164; 165] or runtime infrastructures [166; 167] that expose information about GUI elements within a GUI. In contrast, pixel-based methods directly analyze the image of a GUI, and thus are non-intrusive and generic. Due to the cross-platform characteristics of pixel-based methods, they can be widely used for novel applications such as robotic testing of touch-screen applications [27], linting of GUI visual effects [53] in both Android and IOS.

Pixel-based recognition of GUI elements in a GUI image can be regarded as a domain-specific object detection task. Object detection is a computer-vision technology that detects instances of semantic objects of a certain class (such as human, building, or car) in digital images and videos. It involves two sub-tasks: *region detection or proposal* - locate the bounding box (bbox for short) (i.e., the smallest rectangle region) that contains an object, and *region classification* - determine the class of the object in the bounding box. Existing object-detection techniques adopt a bottom-up strategy: starts with primitive shapes and regions (e.g., edges or contours) and aggregate them progressively into objects. Old-fashioned techniques [33; 34; 168] relies on image features and aggregation heuristics generated by expert knowledge, while deep learning techniques [84; 85; 86] use neural networks to learn to extract features and their aggregation rules from large image data.

GUI elements can be broadly divided into text elements and non-text elements (see Figure 6.1 for the examples of Android GUI elements). Both old-fashioned techniques and deep learning models have been applied for GUI element detection [33; 80; 34; 28; 51]. As detailed in Section 6.2.1, considering the image characteristics of GUIs and GUI elements, the high accuracy requirement of GUI-element region detection, and the design rationale of existing object detection methods, we raise a set of research questions regarding the effectiveness features and models originally designed for generic object detection on GUI elements, the region detection accuracy of statistical machine learning models, the impact of model architectures, hyperparameter settings and training data, and the appropriate ways of detecting text and non-text elements.

These research questions have not been systematically studied. First, existing studies [33; 34; 28] evaluate the accuracy of GUI element detection by only a very small number (dozens to hundreds) of GUIs. The only large-scale evaluation is GUI component design gallery [51], but it tests only the default anchor-box setting (i.e. a predefined set of bboxes) of Faster RCNN [84] (a two-stage model). Second, none of existing studies (including [51]) have investigated the impact of training data size and anchor-box setting on the performance of deep learning object detection models. Furthermore, the latest development of anchor-free object detection has never been attempted. Third, no studies have compared the performance of different methods, for example old fashioned versus deep learning, or different styles of deep learning (e.g., two stage versus one stage, anchor box or free). Fourth, GUI text is simply treated by Optical Character Recognition (OCR) techniques, despite the significant difference between GUI text and document text that OCR is designed for.

To answer the raised research questions, we conduct the first large-scale, comprehensive empirical study of GUI element detection methods, involving a dataset of 50,524 GUI screenshots extracted from 8,018 Android mobile applications (see Section 6.3.2.1), and two representative old-fashioned methods (REMAUI [33] and Xianyu [80]) and three deep learning models (Faster RCNN [84], YOLOv3 [85] and CenterNet [86]) that cover all major method styles (see Section 6.3.2.2). Old-fashioned detection methods perform poorly (REMAUI F1=0.201 and Xianyu F1=0.154 at IoU>0.9) for non-text GUI element detection, which indicates that edge/contour features designed for physical-world objects are not effective for GUI elements. IoU is the intersection area over union area of the detected bounding box and the ground-truth box. Deep learning methods perform much better than old-fashioned methods, and the two-stage anchor-box based Faster RCNN performs the best (F1=0.438 at IoU>0.9), and demands less training data. However, even Faster RCNN cannot achieve a good balance of the coverage of the GUI elements and the accuracy of the detected bounding boxes. Compared with other deep learning methods (one stage anchor-box based YOLO and anchor-free CenterNet), Faster RCNN demands less training data.

It is surprising that anchor-box based models are robust to the anchor-box settings, and merging the detection results by different anchor-box settings can improve the final performance. Our study shows that detecting text and non-text GUI elements by a single model performs much worse than by a dedicated text and non-text model respectively. GUI text should be treated as scene text rather than document text, and the state-of-the-art deep learning scene text model EAST [169] (pretrained without fine tuning) can accurately detect GUI text.

Inspired by these findings, we design a novel approach for GUI element detection, i.e., detect text and non-text elements separately. For non-text GUI element detection, we adopt the simple two-stage architecture: perform region detection and region classification in a pipeline. For non-text region detection, we prefer the simplicity and the bounding-box accuracy of old-fashioned methods, because it does not require training and parameter optimization. By taking into account the unique boundary, shape, texture and layout characteristics of GUI elements, we design a novel old-fashioned method with a top-down coarse-to-fine detection strategy, rather than the current bottom-up edge/contour aggregation strategy in existing methods [33; 80]. Our experiments show that our new old-fashioned non-text region detection outperform the best performing model Faster RCNN, with much higher bounding-box accuracy and GUI element coverage. For non-text region classification and GUI text detection, we adopt the mature, easy-to-deploy ResNet50 image classifier [138] and the EAST scene text detector [169], respectively. By a synergy of our novel old-fashioned methods and existing mature deep learning models, our new method achieves 0.573 in F1 for all GUI elements, 0.523 in F1 for non-text GUI elements, and 0.516 in F1 for text elements in a large-scale evaluation with 25,000 GUI images, which significantly outperform existing old-fashioned methods, and outperform the best deep learning model by 19.4% increase in F1 for non-text elements and 47.7% increase in F1 for all GUI elements.

This chapter contains the following contributions:

- We perform the first systematic analysis of the problem scope and solution space of GUI element detection, and identify the key challenges to be addressed, the limitations of existing solutions, and a set of unanswered research questions.
- We conduct the first large-scale empirical study of seven representative GUI element detection methods, which systematically answers the unanswered questions. We identify the pros and cons of existing methods which informs the design of new methods for GUI element detection.
- We develop a novel approach that effectively incorporates the advantages of different methods and achieves the state-of-the-art performance in GUI element detection.

6.2 Problem Scope and Solution Space

In this section, we identify the unique characteristics of GUIs and GUI elements, which have been largely overlooked when designing or choosing GUI element detection methods (Section 6.2.1). We also summarize representative methods for GUI

Table 6.1: Existing solutions for non-text GUI element detection and their limitations - Old fashioned techniques

Method	Edge/contour aggregation [33; 80; 34]	Template matching [81; 27; 26; 30]
Region Detection	• Detect primitive edges and/or regions, and merge them into larger regions (windows or objects) • Merge with text regions recognized by OCR • Ineffective for artificial GUI elements (e.g., images)	• Depend on manual feature engineering (either sample images or abstract prototypes) • Match samples/prototypes to detect object bounding box and class at the same time • Only applicable to simple and standard GUI elements (e.g., button, checkbox) • Hard to apply to GUI elements with large variance of visual features
Region Classification	Heuristically distinguish image, text, list, container [33; 80]. Can be enhanced by a CNN classification like in [34]	

Table 6.2: Existing solutions for non-text GUI element detection and their limitations - Deep learning techniques

Method	Anchor-box, two stage [84; 51]	Anchor-box, one stage [85; 28]	Anchor free [86]
Region Detection	• Must define anchor boxes • Pipeline region detection and region classification • Gallery D.C. [51] is the only work that tests the Faster RCNN on large-scale real GUIs, but it uses default settings	• YOLOv2 [87] and YOLOv3 [85] uses k-means to determine anchor boxes (k is user-defined) • Simultaneously region detection and region classification • [28] uses YOLOv2; trains and tests on artificial desktop GUIs; only tests on 250 real GUIs	Never applied
Region Classification	A CNN classifier for region classification, trained jointly with region proposal network		

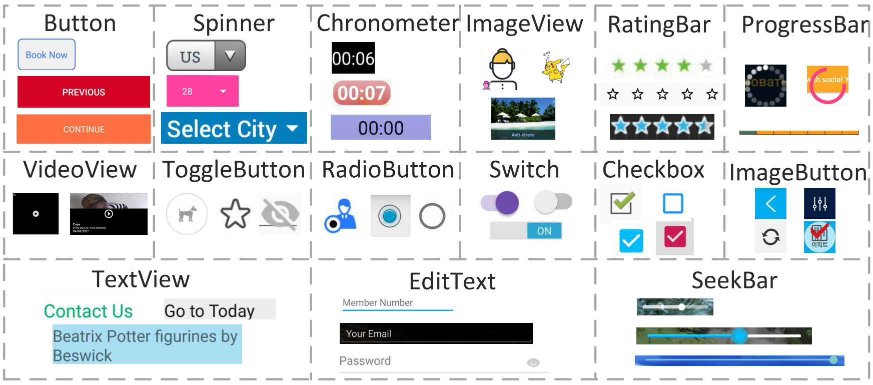


Figure 6.1: Characteristics of GUI elements: large in-class variance and high cross-class similarity

element detection and point out the challenges that the unique characteristics of GUIs and GUI elements pose to these methods (Section 6.2.2).

6.2.1 Problem Scope

Figure 6.1 and Figure 6.6 shows examples of GUI elements and GUIs in our dataset. We observe two element-level characteristics: large in-class variance and high cross-class similarity, and two GUI-level characteristics: packed scene and close-by elements, and mix of heterogeneous objects. In face of these characteristics, GUI element detection must achieve high accuracy on region detection.

Large in-class variance: GUI elements are artificially designed, and their properties (e.g., height, width, aspect ratio and textures) depend on the content to display, the interaction to support and the overall GUI designs. For example, the width of Button or EditText depends on the length of displayed texts. ProgressBar may have different styles (vertical, horizontal or circle). ImageView can display images with any objects or contents. Furthermore, different designers may use different texts, colors, backgrounds and look-and-feel, even for the same GUI functionality. In contrast, physical-world objects, such as human, car or building, share many shape, appearance and physical constraints in common within one class. Large in-class variance of GUI elements pose main challenge of accurate region detection of GUI elements.

High cross-class similarity: GUI elements of different classes often have similar size, shape and visual features. For example, Button, Spinner and Chronometer all have rectangle shape with some text in the middle. Both SeekBar and horizontal ProgressBar show a bar with two different portions. The visual differences to distinguish different classes of GUI elements can be subtle. For example, the difference between Button and Spinner lies in a small triangle at the right side of Spinner, while a thin underline distinguishes EditText from TextView. Small widgets are differentiated by small visual cues. Existing object detection tasks usually deal with physical objects with distinct features across classes, for example, horses, trucks, persons and birds in the popular COCO2015 dataset [170]. High cross-class similarity affects not only region classification but also region detection by deep learning models, as these two subtasks are jointly trained.

Mix of heterogeneous objects: GUIs display widgets, images and texts. Widgets are artificially rendered objects. As discussed above, they have large in-class variance and high cross-class similarity. ImageView has simple rectangle shape but can display any contents and objects. For the GUI element detection task, we want to detect the ImageViews themselves, but not any objects in the images. However, the use of visual features designed for physical objects (e.g., canny edge [82], contour map [83]) contradicts this goal. In Figure 6.6 and Figure 6.4, we can observe a key difference between GUI texts and general document texts. That is, GUI texts are often highly cluttered with the background and close to other GUI elements, which pose main challenge of accurate text detection. These heterogeneous properties of GUI elements must be taken into account when designing GUI element detection methods.

Packed scene and close-by elements: As seen in Figure 6.6, GUIs, especially those of mobile applications, are often packed with many GUI elements, covering almost all the screen space. In our dataset (see Section 6.3.2.1), 77% of GUIs contain more than seven GUI elements. Furthermore, GUI elements are often placed close side by side and separated by only small padding in between. In contrast, there are only an average of seven objects placed sparsely in an image in the popular COCO(2015) object detection challenge [170]. GUI images can be regarded as packed scenes. Detecting objects in packed scenes is still a challenging task, because close-by objects interfere the accurate detection of each object's bounding box.

High accuracy of region detection For generic object detection, a typical correct detection is defined loosely, e.g., by an IoU>0.5 between the detected bounding box and its ground truth (e.g., the PASCAL VOC Challenge standard [171]), since people can recognize an object easily from major part of it. In contrast, GUI element detection has a much stricter requirement on the accuracy of region detection. Inaccurate region detection may not only result in inaccurate region classification, but more importantly it also significantly affects the downstream applications, for example, resulting in incorrect layout of generated GUI code, or clicking on the background in vain during GUI testing. However, the above GUI characteristics make the accurate region detection a challenging task. Note that accurate region classification is also important, but the difficulty level of region classification relies largely on the downstream applications. It can be as simple as predicting if a region is tapable or editable for GUI testing, or if a region is a widget, image or text in order to wireframe a GUI, or which of dozens of GUI framework component(s) can be used to implement the region.

6.2.2 Solution Space

We summarize representative methods for GUI element detection, and raise questions that have not been systematically answered.

6.2.2.1 Non-Text Element Detection

Table 6.2 and Table 6.1 summarize existing methods for non-text GUI element detection. By contrasting these methods and the GUI characteristics in Section 6.2.1, we raise a series questions for designing effective GUI element detection methods. We focus our discussion on region detection, which aims to distinguish GUI element regions from the background. Region classification can be well supported by a CNN-based image classifier [34].

The effectiveness of physical-world visual features. Old-fashioned methods for non-text GUI element detection rely on either edge/contour aggregation [33; 80; 34] or template matching [81; 27; 26; 30]. Canny edge [82] and contour map [83] are primitive visual features of physical-world objects, which are designed to capture fine-grained texture details of objects. However, they do not intuitively correspond to the shape and composition of GUI elements. It is error-prone to aggregate these fine-grained regions into GUI elements, especially when GUIs contain images with physical-world objects. Template matching methods improve over edge/contour aggregation by guiding the region detection and aggregation with high-quality sample images or abstract prototypes of GUI elements. But this improvement comes with the high cost of manual feature engineering. As such, it is only applicable to simple and standard GUI widgets (e.g., button and checkbox of desktop applications). It is hard to apply template-matching method to GUI elements of mobile applications which have large variance of visual features. Deep learning models [84; 51; 85; 28; 86] remove the need of manual feature engineering by learning GUI element features and their composition from large numbers of GUIs. *How effective can deep learning models learn GUI element features and their composition in face of the unique characteristics of GUIs and GUI elements?*

The accuracy of bounding box regression. Deep learning based object detection learns a statistical regression model to predict the bounding box of an object. This regression model makes the prediction in the feature map of a high layer of the CNN, where one pixel stands for a pixel block in the original image. *Can such statistical regression satisfy the high-accuracy requirement of region detection, in face of large in-class variance of GUI element and packed or close-by GUI elements?*

The impact of model architectures, hyperparameters and training data. Faster RCNN [84] and YOLOv2 [85]) have been applied to GUI element detection. These two models rely on a set of pre-defined anchor boxes. The number of anchor boxes and their height, width and aspect ratio are all the model hyperparameters, which are either determined heuristically [84] or by clustering the training images using k-means and then using the metrics of the centroid images [85] Considering large in-class variance of GUI elements, *how sensitive are these anchor-box based models to the definition of anchor boxes, when they are applied to GUI element detection?* Furthermore, the recently proposed anchor-free model (e.g., CenterNet [86]) removes the need of pre-defined anchor-boxes, but has never been applied to GUI element detection. *Can anchor-free model better deal with large in-class variance of GUI elements?* Last but not least, the performance of deep learning models heavily depends on sufficient training data. *How well these models perform with different amount of training data?*

6.2.2.2 Text Element Detection

Existing methods either do not detect GUI texts or detect GUI texts separately from non-text GUI element detection. They simply use off-the-shelf OCR tools (e.g., Tesseract [172]) for GUI text detection. OCR tools are designed for recognizing texts in document images, but GUI texts are very different from document texts. *Is OCR really appropriate for detecting GUI texts? Considering the cluttered background of GUI texts, would it better to consider GUI text as scene text? Can the deep learning scene text model effectively detect GUI texts?* Finally, considering the heterogeneity of GUI widgets, images and texts, *can a single model effectively detect text and non-text elements?*

6.3 Empirical Study of Existing Object Detection Models

To answer the above unanswered questions, we conduct the first large-scale empirical study of using both old-fashioned and deep learning methods for GUI element detection. Our study is done on a dataset of 50,524 GUI screenshots from the Rico dataset [32], which were extracted from 8,018 Android mobile applications from 27 application categories. Our study involves a systematic comparison of two old-fashioned methods, including the representative method REMAUI [33] in the literature and the method Xianyu [80] recently developed by the industry, and three popular deep learning methods that cover all major model design styles, including two anchor-box based methods - Faster RCNN [84] (two stage style) and YOLO V3 [85] (one stage style) and one one-stage anchor-free model CenterNet [86]. For GUI text detection, we compare OCR tool Tesseract [172] and scene text detector EAST [169], and compare separate and unified detection of text and non-text GUI elements.

6.3.1 Research Questions

As region classification can be well supported by a CNN-based image classifier [34], the study focuses on three research questions (RQs) on region detection in GUI element detection task:

- **RQ1 Performance**: How effective can different methods detect the region of non-text GUI elements, in terms of the accuracy of predicted bounding boxes and the coverage of GUI elements?

- **RQ2 Sensitivity**: How sensitive are deep learning techniques to anchor-box settings and amount of training data?

- **RQ3 Text detection**: Does scene text recognition fit better for GUI text detection than OCR technique? Which option, separated versus unified text and non-text detection, is more appropriate?

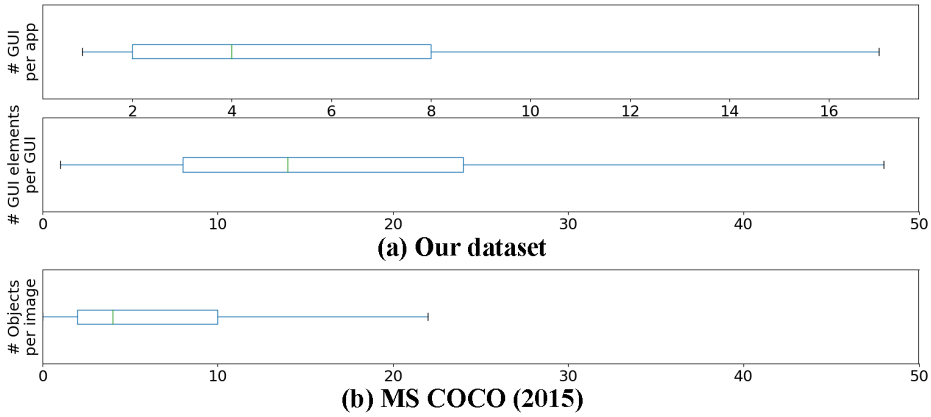


Figure 6.2: GUI elements distribution in our dataset

6.3.2 Experiment Setup

6.3.2.1 Dataset

We leverage Rico dataset [32] to construct our experimental dataset. In this study, we consider 15 types of commonly used GUI elements in the Android Platform (see examples in Figure 6.1). The Rico dataset contains 66,261 GUIs, and we filter out 15,737 GUIs. Among them, 5,327 GUIs do not belong to the app itself, they are captured outside the app, such as Android home screen, a redirection to the social media login page (e.g. Facebook) or to a browser. We identify them by checking whether the package name in the metadata for each GUI is different from the app's true package name. 2,066 GUIs do not have useful metadata, which only contain elements that describe the layout, or elements with invalid bounds, or do not have visible leaf elements. 709 of them do not contain any of the 15 elements. The rest 7,635 GUIs are removed because they only contain text elements or non-text elements. To avoid potential noise/bias, we also remove them. As a result, we obtain 50,524 GUI screenshots that contain at least one of these 15 types of GUI elements. These GUIs are from 8,018 Android mobile applications of 27 categories. These GUIs contain 923,404 GUI elements, among which 426,404 are non-text elements and 497,000 are text elements. We remove the standard OS status and navigation bars from all GUI screenshots as they are not part of application GUIs. We obtain the bounding-box and class of the GUI elements from the corresponding GUI metadata. Figure 6.2 shows the distribution of GUIs per application and the number of GUI elements per GUI. Compared with the number of objects per image in COCO2015, our GUI images are much more packed. We split these 50,524 GUIs into train/validation/test dataset with a ratio of 8:1:1 (40K:5K:5k). All GUIs of an application will be in only one split to avoid the bias of "seen samples" across training, validation and testing. Due to the GPU limitation and the large number of experiments to run, we perform 5-fold cross-validation in all the experiments.

6.3.2.2 Baselines

The baseline methods used in this study include:

REMAUI [33] detects GUI elements with a bottom-up strategy, and detects text and non-text elements separately. For text elements, it uses the OCR tool Tesseract [172], and filters out some false positive text regions based on some predefined heuristics regarding width, height, area and the content. For non-text elements, it detects the structural edge of GUI elements using Canny edge detection [82]. REMAUI then performs edge merging, obtains the contours and the bounding box of the GUI elements by merging partial overlapping regions. Finally, the text and non-text regions are merged based on some predefined rules to obtain a set of GUI elements. We use REMAUI tool [173] provided by its authors in our experiments.

Xianyu [80] is a tool developed by the Alibaba to generate code from GUI images. We only use the element detection part of this tool. Xianyu binarizes the image and performs horizontal/vertical slicing, i.e., cutting the whole images horizontally/vertically in half, recursively to obtain the GUI elements. It uses Laplacian Edge Detection to detect edges and contours in the binarized image and applies flood fill algorithm [174] to identify the connected regions and remove noises from complex background.

Faster RCNN [84] is a two-stage anchor-box-based deep learning technique for object detection. It first generates a set of region proposals by a region proposal network (RPN), also called as region of interests (RoIs), which likely contain objects. In particular, it feeds the image into the convolutional neural network (CNN) to extract the feature map, then uses the region proposal network (RPN) to generate the RoIs. RPN uses a fixed set of user-defined boxes with different scales and aspect ratios (called anchor boxes) and computes these anchor boxes in each point in the feature map. For each box, RPN then computes an objectness score to determine whether it contains an object or not, and regresses it to fit the actual bounding box of the contained object. The second stage is a CNN-based image classifier that determines the object class in the RoIs.

YOLOv3 [85] is an one-stage anchor-box-based object detection technique. Different from the manually-defined anchor box of Faster-RCNN, YOLOv3 uses k-means method to cluster the ground truth bounding boxes in the training dataset, and takes the box scale and aspect ratio of the k centroids as the anchor boxes. It also extracts image features using CNN, and for each grid of the feature map, it generates a set of bounding boxes. For each box, it computes the objectness scores, regresses the box coordinates and classifies the object in the bounding box.

CenterNet [86] is an one-stage anchor-free object detection technique. Instead of generating bounding box based on the predefined anchor boxes, it predicts the position of the top-left and bottom-right corners and the center of an object, and then assembles them to get the bounding box of an object. It matches top-left corners with bottom-right corners if their distance is less than a threshold, and only keep pairs whose center point has a centerness score higher than a threshold.

Tesseract [172] is an OCR tool for document texts. It consists of two steps: text line detection and text recognition. Only the text line detection is relevant to our study. The Tesseract's text line detection is old-fashioned. It first converts the image into binary map, and then performs a connected component analysis to find the outlines of the elements. These outlines are then grouped into blobs, which are further merged together. Finally, it merges text lines that overlap at least half horizontally.

EAST [169] is a deep learning technique to detect text in natural scenes. An input image is first fed into a feature pyramid network. EAST then computes six values for each point based on the final feature map, namely, an objectness score, top/left/bottom/right offsets and a rotation angle. For this baseline, we directly use the pre-trained model to detect texts in GUIs without any fine-tuning.

6.3.2.3 Model Training

For Faster RCNN, YOLOv3 and CenterNet, we initialize their parameters using the corresponding pretrained models of COCO object detection dataset and finetune all parameters using our GUI training dataset. We train each model for 160 iterations with a batch size of 8, and use Adam optimizer. Faster RCNN uses ResNet-101[138] as the backbone. YOLOv3 uses Darknet-53[85] as the backbone. CenterNet uses Hourglass-52[175] as the backbone. For Xianyu and REMAUI, we perform the parameter tuning and use the best setting in all our evaluation. We perform non-maximum suppression (NMS) to remove highly-duplicated predictions in all experiments. It keeps the prediction with the highest objectness in the results and removes others that have a IoU with the selected one over a certain value. We find the best object confidence threshold for each model using the validation dataset.[1]

6.3.2.4 Metrics

For region detection evaluation, we ignore the class prediction results and only evaluate the ability of different methods to detect the bounding box of GUI elements. We use precision, recall and F1-score to measure the performance of region detection. Precision is $TP/(TP+FP)$ and recall is $TP/(TP+FN)$. True positive (TP) refers to a detected bounding box which matches a ground truth box. False positive (FP) refers to a detected box which does not match any ground truth boxes. False negative (FN) refers to a ground truth bounding box which is not matched by any detected boxes. We compute F1-score as: $F1 = (2 \times Precision \times Recall)/(Precision + Recall)$. TP is determined based on the Intersection over Union (IoU) of the two boxes. IoU is calculated by dividing the intersection area I of the two boxes A and B by the union area of the two boxes, i.e., $I/(A+B-I)$. A detected box is considered as a TP if the highest IoU of this box with any ground-truth boxes in the input GUI image is higher than a predefined IoU threshold. Each ground truth box can only be matched at most once and NMS technique is used to determine the optimal matching results. Considering the high accuracy requirement of GUI element detection, we take the IoU threshold 0.9 in most of our experiments.

[1] All codes and models are released at our GitHub repository.

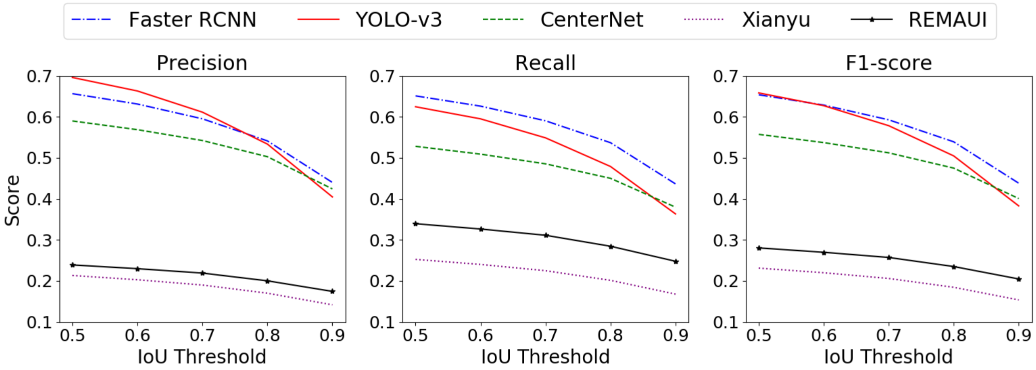


Figure 6.3: Performance at different IoU thresholds

6.3.3 Results - RQ1 Performance

This section reports the performance of five methods for detecting the regions of non-text GUI elements in a GUI. In this RQ, Faster RCNN uses the customized anchor-box setting and YOLOv3 uses k=9 (see Section 6.3.4.1).

6.3.3.1 Trade-off between Bounding-Box Accuracy and GUI-Element Coverage

Figure 6.3 shows the performance of five methods at different IoU thresholds. The F1-score of all deep learning models drop significantly when the IoU threshold increases from 0.5 to 0.9, with the 31%, 45% and 28% decrease for Faster-RCNN, YOLOv3 and CenterNet respectively. The bounding box of a RoI is predicted by statistical regression in the high-layer feature map of the CNN, where one pixel in this abstract feature map corresponds to a pixel block in the original image. That is, a minor change of the predicted coordinates in the abstract feature map will lead to a large change in the exact position in the original image. Therefore, deep learning models either detect more elements with loose bounding boxes or detect less elements with accurate bounding boxes. In contrast, the F1-score of REMAUI and Xianyu does not drop as significantly as that of deep learning models as the IoU threshold increases, but their F1-scores are much lower than those of deep learning models. This suggests that the detected element regions by these old-fashioned methods are mostly noise, but when they do locate real elements, the detected bounding boxes are fairly accurate.

6.3.3.2 Performance Comparison

We observe that if the detected bounding box has <0.9 IoU over the corresponding GUI element, not only does the box miss some portions of this element, but it also includes some portions of adjacent elements due to the packed characteristic of GUI design. Therefore, we use IoU> 0.9 as an acceptable accuracy of bounding box prediction. Table 6.3 shows the overall performance of the five methods at IoU>0.9 threshold for detecting non-text GUI elements.

Table 6.3: Performance: non-text element detection (IoU>0.9)

Method	#bbox	Precision	Recall	F1
REMAUI	54,903	0.175	0.238	0.201
Xianyu	47,666	0.142	0.168	0.154
Faster-RCNN	39,995	**0.440**	**0.437**	**0.438**
YOLOv3	36,191	0.405	0.363	0.383
CenterNet	36,096	0.424	0.380	0.401

Xianyu performs the worst, with all metrics below 0.17. We observe that Xianyu works fine for simple GUIs, containing some GUI elements on a clear or gradient background (e.g., Xianyu-(c)/(d) in Figure 6.6). When the GUI elements are close-by or placed on a complex background image, Xianyu's slicing method and its background de-noising algorithms do not work well. For example, in Xianyu-(a)/(b) in Figure 6.6, it misses most of GUI elements. Xianyu performs slicing by the horizontal or vertical lines across the whole GUI. Such lines often do not exist in GUIs, especially when they have complex background images (Xianyu-(a)) or the GUI elements are very close-by (Xianyu-(b)). This results in many under-segmentation of GUI images and the misses of many GUI elements. Furthermore, Xianyu sometimes may over-segment the background image (Xianyu-(a)), resulting in many noise non-GUI-element regions.

REMAUI performs better than Xianyu, but it is still much worse than deep learning models. It suffers from similar problems as Xianyu, including ineffective background de-noising and over-segmentation. It outperforms Xianyu because it merges close-by edges to construct bounding boxes instead of the simple slicing method by horizontal/vertical lines. However, for GUIs with image background, its edge merging heuristics often fail due to the noisy edges of physical-world objects in the images. As such, it often reports some non-GUI-element regions of the image as element regions, or erroneously merges close-by elements, as shown in Figure 6.6. Furthermore, REMAUI merges text and non-text region heuristically, which are not very reliable either (see the text elements detected as non-text elements in REMAUI-(b)/(c)/(d)).

Deep learning models perform much better than old fashioned methods. In Figure 6.6, we see that they all locate some GUI elements, even those overlaying on the background picture. These models are trained with large-scale data, "see" many sophisticated GUIs, and thus can locate GUI elements even in a noisy background. However, we also observe that the detected bounding boxes by deep learning models may not be very accurate, as they are estimated by a statistical regression model. The two-stage model Faster RCNN outperforms the other two one-stage models YOLOv3 and CenterNet. As discussed in Section 6.2.1, GUI elements have large in-class variance and high cross-class similarity. Two stage models perform region detection and region classification in a pipeline so that these two steps are less mutually interfered,while one-stage models perform region detection and region classification simultaneously.

Table 6.4: Impact of anchor-box settings (IoU>0.9)

Setting	Precision	Recall	F1
Faster RCNN Default	0.433	0.410	0.421
Faster RCNN Customized	0.440	0.437	0.438
Faster RCNN Union	0.394	0.469	0.428
Faster RCNN Intersection	**0.452**	**0.460**	**0.456**
YOLOv3 k=5	0.394	0.333	0.361
YOLOv3 k=9	0.405	0.363	0.383
YOLO Union	0.372	0.375	0.373
YOLO Intersection	**0.430**	**0.424**	**0.427**

Between the two one-stage models, anchor-free CenterNet outperforms anchor-box-based YOLOv3 at IoU>0.9. However, YOLOv3 performs better than CenterNet at lower IoU thresholds (see Figure 6.3). Anchor-free model is flexible to handle the large in-class variance of GUI elements and GUI texts (see more experiments on GUI text detection in Section 6.3.5). However, as shown in Figure 6.6, this flexibility is a double-blade, which may lead to less accurate bounding boxes, or bound several elements in one box (e.g., CenterNet-(a)/(d)). Because GUI elements are often close-by or packed in a GUI, CenterNet very likely assembles the top-left and bottom-right corners of different GUI elements together, which leads to the wrong bounding boxes.

Deep learning models significantly outperform old-fashioned detection methods. Two-stage anchor-box-based models perform the best in non-text GUI element detection task. But it is challenging for the deep learning models to achieve a good balance between the accuracy of the detected bounding boxes and the detected GUI elements, especially for anchor-free models.

6.3.4 Results - RQ2 Sensitivity

This section reports the sensitivity analysis of the deep learning models for region detection from two aspects: anchor-box settings and amount of training data.

6.3.4.1 Anchor-Box Settings

For Faster RCNN, we use two settings: the default setting (three anchor-box scales - 128, 256 and 512, and three aspect ratios - 1:1, 1:2 and 2:1); and the customized setting (five anchor-box scales - 32, 64, 128, 256 and 512, and four aspect (width:height) ratios - 1:1, 2:1, 4:1 and 8:1). This customized setting is drawn from the frequent scales and aspect ratios of the GUI elements in our dataset. Considering the size of GUI elements, we add two small scales 32 and 64. Furthermore, we add two more aspect ratios to accommodate the large variance of GUI elements. For YOLOv3, we use two *k* settings: 5 and 9, which are commonly used in the literature. YOLOv3 automatically derives anchor-box metrics from *k* clusters of GUI images in the dataset.

Table 6.5: Impact of amount of training data (IoU>0.9)

Method	Size	Precision	Recall	F1
Faster-RCNN	2K	0.361	0.305	0.331
	10K	0.403	0.393	0.398
	40K	**0.440**	**0.437**	**0.438**
YOLOv3	2K	0.303	0.235	0.265
	10K	0.337	0.293	0.313
	40K	**0.405**	**0.363**	**0.383**
CenterNet	2K	0.319	0.313	0.316
	10K	0.328	0.329	0.329
	40K	**0.424**	**0.380**	**0.401**

Table 6.4 shows the model performance (at IoU>0.9) of these different anchor-box settings. It is somehow surprising that there is only a small increase in F1 when we use more anchor-box scales and aspect ratios. We further compare the TPs of different anchor-box settings. We find that 55% of TPs overlap between the two settings for Faster RCNN, and 67% of TPs overlap between the two settings for YOLOv3. As the scales and aspect ratios of GUI elements follow standard distributions, using a smaller number of anchor boxes can still cover a large portion of the element distribution.

As different settings detect some different bounding boxes, we want to see if the differences may complement each other. To that end, we adopt two strategies to merge the detected boxes by the two settings: union strategy and intersection strategy. For two overlapped boxes, we take the maximum objectness of them, and then merge the two boxes by taking the union/intersection area for union/interaction strategy. For the rest of the boxes, we directly keep them. We find the best object confidence threshold for the combined results using the validation dataset. The union strategy does not significantly affect the F1, which means that making the bounding boxes larger is not very useful. In fact, for the boxes which are originally TPs by one setting, the enlarged box could even become FPs. However, the intersection strategy can boost the performance of both Faster RCNN and YOLOv3, achieving 0.456 and 0.427 in F1 respectively. It is reasonable because the intersection area is confirmed by the two settings, and thus more accurate.

6.3.4.2 Amount of Training Data.

In this experiment, Faster RCNN uses the customized anchor-box setting and YOLOv3 uses k=9. We train the models with 2K, 10K, 40K training data separately, and test the models on the same 5k GUI images. Each 2k- or 10k experiment uses randomly selected 2k or 10k GUIs in the 40k training data. As shown in Table 6.5, the performance of all models drops as the training data decreases. This is reasonable because deep learning models cannot effectively learn the essential features of the GUI elements without sufficient training data. The relative performance of the three models

Table 6.6: Impact of GUI-element location (2k data)

Method	Location	Precision	Recall	F1
Faster-RCNN	**Top**	**0.505**	**0.453**	**0.477**
	Middle	0.324	0.268	0.293
	Bottom	0.288	0.239	0.261
YOLOv3	**Top**	**0.477**	**0.432**	**0.453**
	Middle	0.240	0.176	0.203
	Bottom	0.247	0.189	0.210
CenterNet	**Top**	**0.396**	**0.422**	**0.409**
	Middle	0.325	0.294	0.309
	Bottom	0.227	0.334	0.270

is consistent at the three training data sizes, with YOLOv3 always being the worst. This indicates the difficulty in training one-stage anchor-box model. Faster RCNN with 2k (or 10k) training data achieves the comparable or higher F1 than that of YOLOv3 and CenterNet with 10k (or 40k) training data. This result further confirms that two-stage model fits better for GUI element detection tasks than one-stage model, and one-stage anchor-free model performs better than one-stage anchor-box model.

6.3.4.3 Location of GUI Elements

An Android mobile application generally has a top app bar and the main content.We also observe that the bottom area of the GUI sometimes has partial elements, which are not fully shown in the current page. In this experiment, we want to see the GUI elements from which parts of the GUIs are most difficult to detect. Based on the observation of the height of the app bar and the bottom partial elements in our dataset, we split a GUI into three parts top:middle:bottom by a height ratio of 15:70:15. A GUI element belongs to a part if its bottom line falls into that part. We train and test the models using the same experiment settings as in Section 6.3.4.2. We still train and test the model on the whole GUI images, but we examine the detection accuracy of the GUI elements in the top, main and bottom parts separately.

Table 6.6 shows the model performance (trained with 2k data) for different parts of GUIs. All the models perform the best for the top part of the GUIs. The top part of the GUIs often contains GUI elements like back button, hamburger menu, etc., which makes the top part similar across different applications. As such, with as little as 2k training data, Faster RCNN, YOLOv3 and CenterNet achieve F1 0.477, 0.453 and 0.409 for the top part of GUIs, respectively. In contrast, the middle and bottom portions of GUIs is much more diverse, and thus the models, especially YOLOv3, cannot learn sufficient features to make accurate detection with just 2k training data. The bottom part seems to be the most difficult part to learn due to the presence of partially shown GUI elements. The much lower F1-scores (around 0.21) of YOLOv3 for the main and bottom parts further confirms the difficulty in effectively training one-stage anchor-box-based model. The results at 10k and 40k training data are consistent with those presented for 2k data. Note that the top part of GUIs usually contains only several GUI elements. Therefore, although the models can make much more accurate detection for the GUI elements in the top part, this does not not significantly boost the overall performance for the whole GUI.

Considering the fact that UIs often have similar items in the top side of the whole UI (e.g, the back button at the top-right position), this feature may enable models with a small training data to achieve relatively good results. To confirm this, we split a UI into three parts from top to bottom in a height ratio of 15:70:15. As seen in Table 6.6, Faster RCNN, YOLO-v3 and CenterNet perform well in the top part of UIs, achieving 0.477, 0.453 and 0.409 in F1 score respectively, but fall short of making a good prediction in the middle and bottom parts of UIs, where elements are of various size, shape, content and style. A small amount of data could not provide enough information for models to filter out noise and learn the essence of data.

Anchor-box settings do not significantly affect the performance of anchor-box-based models, because a small number of anchor boxes can cover the majority of GUI elements. Two-stage anchor-box-based model is the easiest to train, which requires one magnitude less training data to achieve comparable performance as one-stage model. One stage anchor-box model is the most difficult to train.

6.3.5 Results - RQ3 Text Detection

6.3.5.1 Separated versus Unified Text/Non-Text Element Detection

All existing works detect GUI text separately from non-text elements. This is intuitive in that GUI text and non-text elements have very different visual features. However, we were wondering if this is a must or text and non-text elements can be reliably detected by a single model. To answer this, we train Faster RCNN, YOLOv3 and CenterNet to detect both text and non-text GUI elements. Faster RCNN uses the customized anchor-box setting and YOLOv3 uses k=9. The model is trained with 40k data and tested on 5k GUI images. In this RQ, both non-text and text elements in GUIs are used for model training and testing.

Table 6.7 shows the results. When trained to detect text and non-text elements together, Faster RCNN still performs the best in terms of detecting non-text elements. But the performance of all three models for detecting non-text elements degrades, compared with the models trained to detect non-text elements only. This indicates that mixing the learning of text and non-text element detection together interfere with the learning of detecting non-text elements. CenterNet performs much better for detecting text elements than Faster RCNN and YOLOv3, which results in the best overall performance for the mixed text and non-text detection. CenterNet is anchor-free, which makes it flexible to handle large variance of text patterns. So it has comparable performance for text and non-text elements. In contrast, anchor-box-based Faster RCNN and YOLOv3 are too rigid to reliably detect text elements. However, the performance of CenterNet in detecting text elements is still poor. Text

Table 6.7: Text detection: separated versus unified processing

Method	Element		Precision	Recall	F1
Faster-RCNN	nontext-only		0.440	0.437	0.438
	mix	nontext	**0.379**	**0.436**	**0.405**
		text	0.275	0.250	0.262
		both	0.351	0.359	0.355
YOLOv3	nontext-only		0.405	0.363	0.383
	mix	non-text	0.325	0.347	0.335
		text	0.319	0.263	0.288
		both	0.355	0.332	0.343
CenterNet	nontext-only		0.424	0.380	0.401
	mix	non-text	0.321	0.397	0.355
		text	**0.416**	**0.319**	**0.361**
		both	**0.391**	**0.385**	**0.388**

Table 6.8: Text detection: OCR versus scene text

Method	Precision	Recall	F1
Tesseract	0.291	0.518	0.372
EAST	**0.402**	**0.720**	**0.516**
REMAUI	0.297	0.489	0.369
Xianyu	0.272	0.481	0.348

elements always have space between words and lines. Due to the presence of these spaces, CenterNet often detects a partial text element or erroneously groups separate text elements as one element when assembling object corners.

6.3.5.2 OCR versus Scene Test Recognition

Since it is not feasible to detect text and non-text GUI elements within a single model, we want to investigate what is the most appropriate method for GUI text detection. All existing works (e.g., REMAUI, Xianyu) simply use OCR tool like Tesseract. We observe that GUI text is more similar to scene text than to document text. Therefore, we adopt a deep learning scene text recognition model EAST for GUI text detection, and compare it with Tesseract. We directly use the pre-trained EAST model without any fine tuning on GUI text.

As shown in Table 6.8, EAST achieves 0.402 in precision, 0.720 in recall and 0.516 in F1, which is significantly higher than Tesseract (0.291 in precision, 0.518 in recall and 0.372 in F1). Both Xianyu and REMAUI perform some post-processing of the Tesseract's OCR results in order to filter out false positives. But it does not significantly change the performance of GUI text detection. As EAST is specifically designed for scene text recognition, its performance is significantly better than using generic object detection models for GUI text detection (see Table 6.7). EAST detects almost all texts in a GUI, including those on the GUI widgets (e.g., the button labels in Figure 6.4(c)). However, those texts on GUI widgets are considered as part of

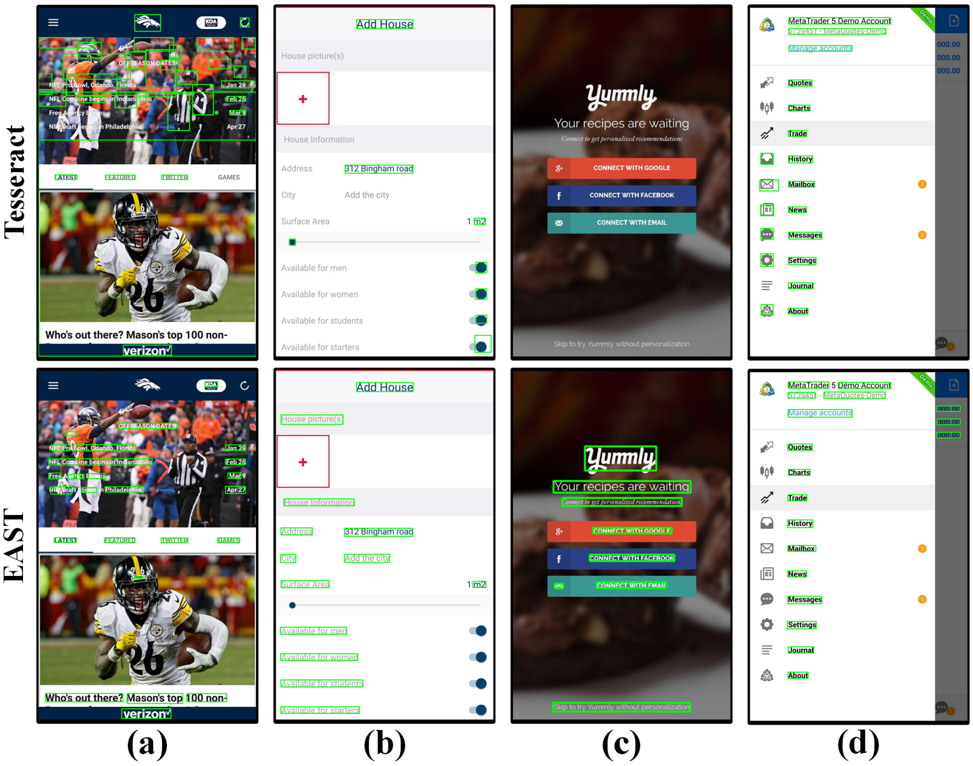


Figure 6.4: Examples: OCR versus scene text

the widgets in our ground-truth data, rather than stand-alone texts. This affects the precision of EAST against our ground-truth data, even though the detected texts are accurate.

Figure 6.4 presents some detection results. Tesseract achieves the comparable results as EAST only for the left side of Figure 6.4(d), where text is shown on a white background just like in a document. From all other detection results, we can observe the clear advantages of treating GUI text as scene text than as document text. First, EAST can accurately detect text in background image (Figure 6.4(a)), while Tesseract outputs many inaccurate boxes in such images. Second, EAST can detect text in a low contrast background (Figure 6.4(b)), while Tesseract often misses such texts. Third, EAST can ignore non-text elements (e.g., the bottom-right switch buttons in Figure 6.4(b), and the icons on the left side of Figure 6.4(d)), while Tesseract often erroneously detects such non-text elements as text elements.

GUI text and non-text elements should be detected separately. Neither OCR techniques nor generic object detection models can reliably detect GUI texts. As GUI texts have the characteristics of scene text, the deep learning scene text recognition model can be used (even without fine-tuning) to accurately detect GUI texts.

6.4 A Novel Approach

Based on the findings in our empirical study, we design a novel approach for GUI element detection. Our approach combines the simplicity of old-fashioned computer vision methods for non-text-element region detection, and the mature, easy-to-deploy deep learning models for region classification and GUI text detection (Section 6.4.1). This synergy achieves the state-of-the-art performance for the GUI element detection task (Section 6.4.2).

6.4.1 Approach Design

Our approach detects non-text GUI elements and GUI texts separately. For GUI text detection, we simply use the pre-trained state-of-the-art scene text detector EAST [169]. For non-text GUI element detection, we adopt the two-stage design, i.e, perform region detection and region classification in a pipeline. For region detection, we develop a novel old-fashioned method with a top-down coarse-to-fine strategy and a set of GUI-specific image processing algorithms. For region classification, we fine-tune the pre-trained ResNet50 image classifier [138] with GUI element images.

6.4.1.1 Region Detection for Non-Text GUI Elements

According to the performance and sensitivity experiments results, we do not want to use generic deep learning object detection models [87; 84; 86]. First, they demand sufficient training data, and different model designs require different scale of training data to achieve stable performance. Furthermore, the model performance is still less optimal even with a large set of training data, and varies across different model designs. Second, the nature of statistical regression based region detection cannot satisfy the high accuracy requirement of GUI element detection. Unlike generic object detection where a typical correct detection is defined loosely (e.g, IoU>0.5) [176]), detecting GUI elements is a fine-grained recognition task which requires a correct detection that covers the full region of the GUI elements as accurate as possible, but the region of non-GUI elements and other close-by GUI elements as little as possible. Unfortunately, neither anchor-box based nor anchor-free models can achieve this objective, because they are either too strict or too flexible in face of large in-class variance of element sizes and texture, high cross-class shape similarity, and the presence of close-by GUI elements.

Unlike deep learning models, old-fashioned methods [33; 80] do not require any training which makes them easy to deploy. Furthermore, when old fashioned methods locate some GUI elements, the detected bounding boxes are usually accurate, which is desirable. Therefore, we adopt old-fashioned methods for non-text GUI-element region detection. However, existing old-fashioned methods use a bottom-up strategy which aggregates the fine details of the objects (e.g., edge or contour) into objects. This bottom-up strategy performs poorly, especially affected by the complex background or objects in the GUIs and GUI elements. As shown in Figure 6.5, our

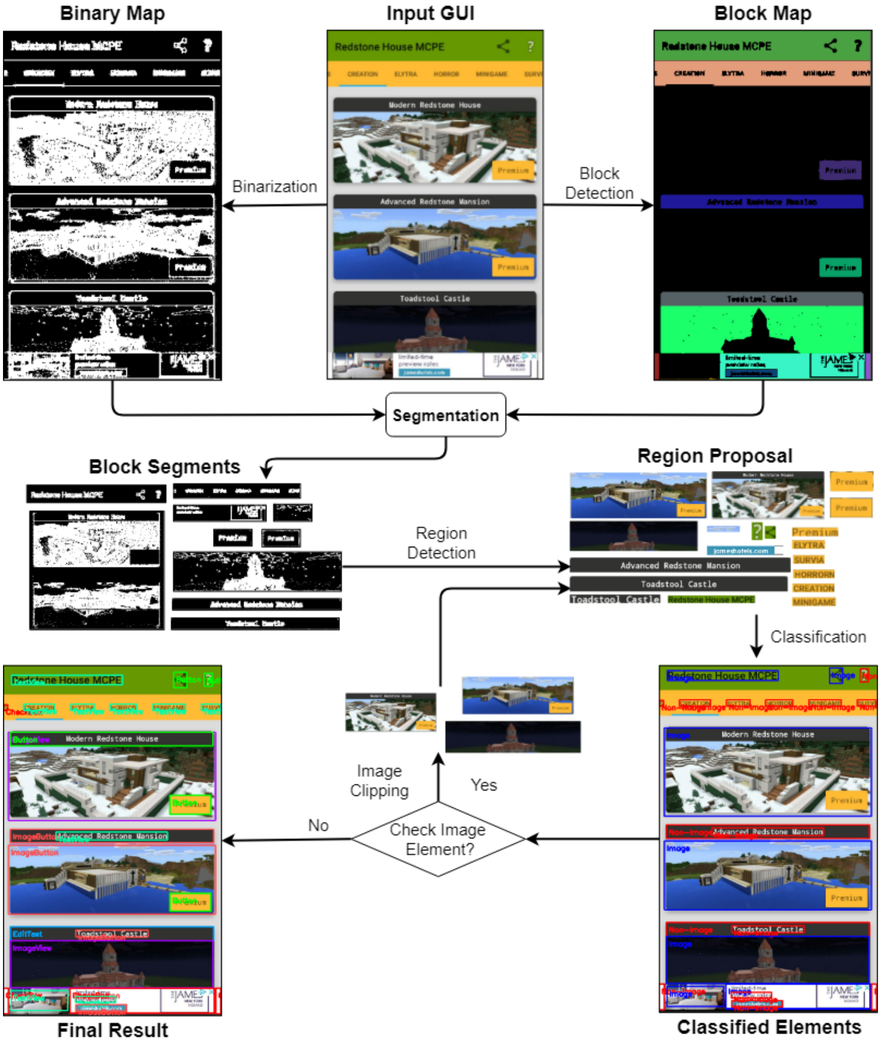


Figure 6.5: Our method for non-text GUI element detection

method adopts a completely different strategy: top-down coarse-to-fine. This design carefully considers the regularity of GUI layouts and GUI-element shapes and boundaries, as well as the significant differences between the shapes and boundaries of artificial GUI elements and those of physical-world objects.

Our region detection method first detects the layout blocks of a GUI. The intuition is that GUIs organize GUI elements into distinct blocks, and these blocks generally have rectangle shape. Xianyu also detects blocks, but it assumes the presence of clear horizontal and vertical lines. Our method does not make this naive assumption. Instead, it first uses the flood-filling algorithm [174] over the grey-scale map of the input GUI to obtain the maximum regions with similar colors, and then uses the shape recognition [177] to determine if a region is a rectangle. Each rectangle region is considered as a block. Finally, it uses the Suzuki's Contour tracing algorithm [83] to compute the boundary of the block and produce a block map. In Figure 6.5, we show the detected block in different colors for the presentation clarity. Note that blocks usually contain some GUI elements, but some blocks may correspond to a particular GUI element.

Next, our method generates a binary map of the input GUI, and for each detected block, it segments the corresponding region of the binary map. Binarization simplifies the input image into a black-white image, on which the foreground GUI elements can be separated from the background. Existing methods [33; 80] perform binarization through Canny edge detection [82] and Sobel edge detection [178], which are designed to keep fine texture details in nature scene images. Unfortunately, this detail-keeping capability contradicts the goal of GUI element detection, which is to detect the shape of GUI elements, rather than their content and texture details. For example, we want to detect an ImageView element no matter what objects are shown in the image (see Figure 6.6). We develop a simple but effective binarization method based on the gradient map [179] of the GUI images. A gradient map captures the change of gradient magnitude between neighboring pixels. If a pixel has small gradient with neighboring pixels, it becomes black on the binary map, otherwise white. As shown in Figure 6.5, the GUI elements stand out from the background in the binary map, either as white region on the black background or black region with white edge.

Our method uses the connected component labeling [180] to identify GUI element regions in each binary block segment. It takes as input the binarized image and performs two-pass scanning to label the connected pixels. For the first scanning, it labels the foreground pixel according to the neighbor points from left to right and from top to bottom. If the current pixel does not have labelled neighbors, it is labeled as 1 and the next pixel with the same situation will be labeled as 2, increased by 1 gradually. If the current pixel have labelled neighbors, it is labeled as the minimum value of the neighbors'. If labelled neighbors have different values, these values are recorded to be in the same group. For the second scanning, the pixels with the label from the same group are grouped together as a connected area. As GUI elements can be any shape, it identifies a smallest rectangle box that covers the detected regions as the bounding boxes. Although our binarization method does not keep many texture details of non-GUI objects, the shape of non-GUI objects (e.g., those buildings in the pictures) may still be present in the binary map. These noisy shapes interfere existing bottom-up aggregation methods [82; 83] for GUI element detection, which results in over-segmentation of GUI elements. In contrast, our top-down detection strategy minimizes the influence of these non-GUI objects, because it uses relaxed grey-scale map to detect large blocks and then uses strict binary map to detect GUI elements. If a block is classified as an image, our method will not further detect GUI elements in this block.

6.4.1.2 Region Classification for Non-Text GUI Elements

For each detected GUI element region in the input GUI, we use a ResNet50 image classifier to predict its element type. In this work, we consider 15 element types

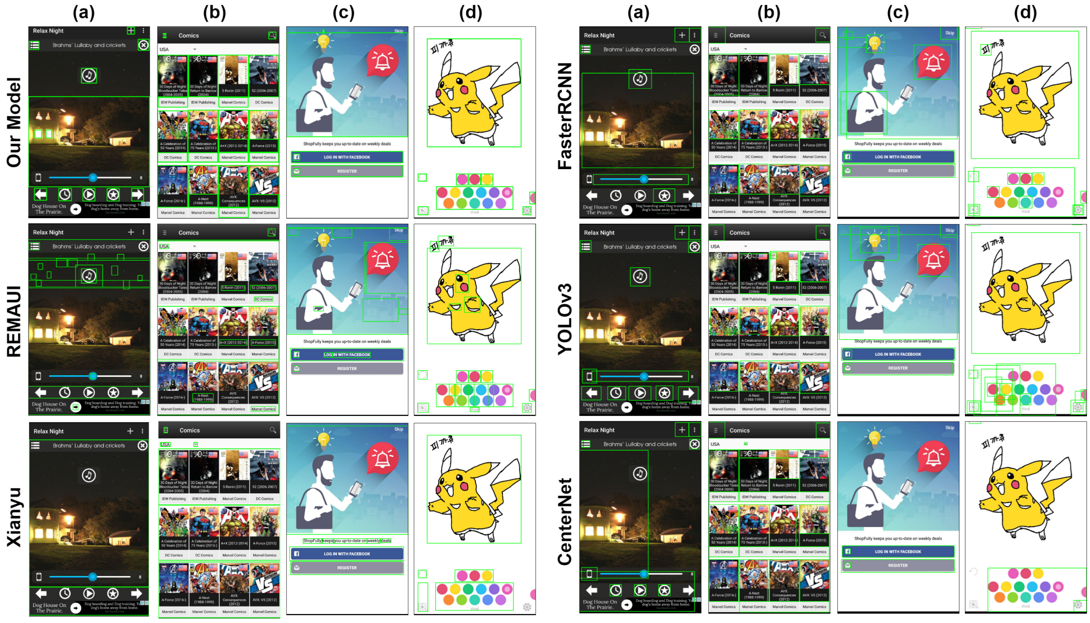


Figure 6.6: Region detection results for non-text GUI element: our method versus five baselines

as shown in Figure 6.1. The Resnet50 image classifier is pre-trained with the ImageNet data. We fine-tune the pre-trained model with 90,000 GUI elements (6,000 per element type) randomly selected from the 40k GUIs in our training dataset.

6.4.1.3 GUI Text Detection

Section 6.3.5 shows that GUI text should be treated as scene text and be processed separately from non-text elements. Furthermore, scene text recognition model performs much better than generic object detection models. Therefore, we use the state-of-the-art deep-learning scene text detector EAST [169] to detect GUI text. As shown in Figure 6.4(c), EAST may detect texts that are part of non-image GUI widgets (e.g., the text on the buttons). Therefore, if the detected GUI text is inside the region of a non-image GUI widgets, we discard this text.

6.4.2 Evaluation

Table 6.9 shows the region-detection performance for non-text, text and both types of elements. For non-text GUI elements, our approach performs better than the best baseline Faster RCNN (0.523 versus 0.438 in F1). For text elements, our approach is overall the same as EAST. It is better than EAST in precision, because our approach discards some detected texts that are a part of GUI widgets. But this degrades the recall. For text and non-text elements as a whole, our approach performs better than the best baseline CenterNet (0.573 versus 0.388 in F1).

Table 6.9: Detection performance of our approach (IoU>0.9)

Elements	Precision	Recall	F1
non-text	0.503	0.545	0.523
text	0.589	0.547	0.516
both	0.539	0.612	0.573

Figure 6.6 shows the examples of the detection results by our approach and the five baselines. Compared with REMAUI and Xianyu, our method detects much more GUI elements and much less noisy non-GUI element regions, because of our robust top-down coarse-to-fine strategy and GUI-specific image processing (e.g., connected component labeling rather than canny edge and contour). Our method also detects more GUI elements than the three deep learning models. Furthermore, it outputs more accurate bounding boxes and less overlapping bounding boxes, because our method performs accurate pixel analysis rather than statistical regression in the high-layer of CNN. Note that deep learning models may detect objects in images as GUI elements, because there are GUI elements of that size and with similar visual features. In contrast, our method detects large blocks that are images and treats such images as whole. As such, our method suffers less over-segmentation problem.

We conclude three main reasons when our model fails. First, same look and feel UI regions may correspond to different types of widgets, such as text label versus text button without border. This is similar to the widget tappability issue studied in [76]. Second, the repetitive regions in a dense UI (e.g., Figure 6.6(b)) often have inconsistent detection results. Third, it is sometimes hard to determine whether a text region is a text label or part of a widget containing text, for example, the spinner showing USA at the top of Figure 6.6(b). Note that these challenges affect all methods. We leave them as our future work.

Table 6.10 shows the region classification results of our CNN classifier and the three deep learning baselines. The results consider only true-positive bounding boxes, i.e., the classification performance given the accurate element regions. As text elements are outputted by EAST directly, we show the results for non-text elements and all elements. We can see that our method outputs more true-positive GUI element regions, and achieves higher classification accuracy (0.86 for non-text elements and 0.91 for all elements, and the other three deep models achieves about 0.68 accuracy). Our classification accuracy is consistent with [34], which confirms that the effectiveness of a pipeline design for GUI element detection.

Table 6.11 shows the overall object detection results, i.e., the true-positive bounding box with the correct region classification over all detected element regions. Among the three baseline models, Faster RCNN performs the best for non-text elements (0.315 in F1), but CenterNet, due to this model flexibility to handle GUI texts, achieves the best performance for all elements (0.282 in F1). Compared with these three baselines, our method achieves much better F1 for both non-text elements (0.449) and all elements (0.524), due to its strong capability in both region detection and region classification.

Table 6.10: Region classification results for TP regions

	Non-text elements		*All elements*	
Method	**#bbox**	**Accuracy**	**#bbox**	**Accuracy**
FasterRCNN	18,577	0.68	34,915	0.68
YOLOv3	15,428	0.64	32,225	0.65
Centernet	16,072	0.68	36,803	0.66
Our method	21,977	**0.86**	53,027	**0.91**

Table 6.11: Overall results of object detection (IoU $>$ 0.9)

	Non-text elements			*All elements*		
Method	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
Faster-RCNN	0.316	0.313	0.315	0.269	0.274	0.271
YOLOv3	0.274	0.246	0.260	0.258	0.242	0.249
CenterNet	0.302	0.270	0.285	0.284	0.280	0.282
Xianyu	0.122	0.145	0.133	0.270	0.405	0.324
REMAUI	0.151	0.205	0.173	0.296	0.449	0.357
Our method	**0.431**	**0.469**	**0.449**	**0.490**	**0.557**	**0.524**

6.5 Threats to Validity

We discuss three types of threats, i.e., internal validity, construct validity and external validity.

6.5.1 Internal Validity

Threats to internal validity concern the unexpected factors that may impact the results. One threat lies in the deep learning models as the performance of the models may change as the training procedure includes some randomness. To mitigate this problem, we performed 5-fold cross-validation in all our experiments and reported the average results. We also release all the code and models.

6.5.2 Construct Validity

Threats to construct validity concern the operationalisation of the experimental artifacts. One potential threat is that we re-implemented the Xianyu tool [80] as they do not provide an open-sourced code. However, they published articles detailing the structure, techniques even the code used in their implementation. We carefully read all their articles and followed what they said. While our implementation may have some slight differences from their original implementation, our version could still represent a typical method for old-fashioned techniques. We also open-source our code in the GitHub repository for researchers to enhance the reproducibility and transparency.

6.5.3 External Validity

Threats to external validity are about generalisability. We trained and evaluated models and our tool using the Rico dataset, which was collected in 2017. This dataset may be outdated as the design keeps changing and more fancy styles emerge. However, no matter how the trend changes, the underlying design principles remain similar, which guarantees the validity of the Rico dataset. We believe even though we tested on the old dataset, our tool and conclusions can be easily generalised to future and updated UIs. Another threat is the generalisability to other platforms. We only evaluated our dataset in Android UIs, without further evaluating the performance in other platforms, like iOS UIs and website UIs. However, our summarised characteristics should be easily generalised to other UIs on other platforms as all UIs share the same basic elements like images and buttons. In addition, our proposed tool does not need to be trained. Unlike the deep learning models, it is screen size agnostic. We believe our tools can be applied to detect and locate UI elements in all platforms, but further evaluation is needed to confirm this.

6.6 Conclusion

This chapter investigates the problem of GUI element detection. We identify four unique characteristics of GUIs and GUI elements, including large in-class variance, high cross-class similarity, packed or close-by elements and mix of heterogeneous objects. These characteristics make it a challenging task for existing methods (no matter old fashioned or deep learning) to accurately detect GUI elements in GUI images. Our empirical study reveals the underperformance of existing methods borrowed from computer vision domain and the underlying reasons, and identifies the effective designs of GUI element detection methods. Informed by our study findings, we design a new GUI element detection approach with both the effective designs of existing methods and the GUI characteristics in mind. Our new method achieves the state-of-the-art performance on the largest-ever evaluation of GUI element detection methods.

Chapter 7

Discussions

In this section, we discuss the limitations of each work and potential future directions to improve them. I also note that after I published these works, there are many following works that mitigate the limitations of each work. I will discuss them in this chapter.

7.1 UI Design Search

Existing research shows that explicitly providing information to models can achieve better performance than implicit input [181]. Therefore, as we implicitly encode the types and coordinates of elements and their relationship in a pure wireframe input, adding explicit input can advance the performance of models. One direction is to explicitly encode the position of each element and their structural relationship to help models better learn the component concept, i.e., a group of elements that are used together for a specific purpose. This improvement can help find designs where the components are in different orders. In our work, WAE may recommend designs that have all input fields on the top half of the screen as the input wireframe also have these layout. Manandhar et al. [182] incorporate this information by using a graph to represent UI designs. They encode the structural and semantic information about the UI using a graph convolutional network, and decode using a CNN. They consider a Siamese network to train the model. Their experiments demonstrate the superior advantage of explicitly including the graph information.

Another line is to consider a multi-modal input to better learn the semantics of each piece of information by separate models. Screen2vec [183], proposed by Li et al., follows this direction. They fully leverage all information in the view hierarchy, including text contents, types and positions of UI elements, and additional app descriptions to learn the semantics of UI designs. They adopt sentence-Bert to encode text features, an RNN model to learn the relationship between components, and an auto-encoder to obtain the layout information. After that, a linear layer is used to merge and integrate these features, and the final features are the concatenation with a high-level app description. The inclusion of explicit features of all information boosts the performance of the UI search engine and gives the engine the capability of recommending UIs with different design styles but sharing the same goal as the query wireframe. Future directions could be including surrounding UIs in interaction traces to better add the usage logic in recommendations. However, all the above techniques require the view hierarchy information, but it is not often the case. Combining the UI element detection tool we proposed [36] or from other works [35] can extend the usage of these techniques. Another limitation is that these research could not guarantee the recommended UI designs are of high quality. Integrating design knowledge, such as Google Material Design, could provide a more intelligent design search engine that acts like a professional designer guiding junior designers. Moreover, all these focus on UI design in Android mobile applications, generalisation to other platforms, like iOS applications, and even desktop applications would be an interesting topic.

Apart from these two directions that aim to improve UI design search, other related works also show the capability of improving the search engine. Wang et al. [184] propose screen2words to generate UI descriptions for each UI given the screenshot, view hierarchy information and app descriptions. Wu et al. [185] instead input the UI elements and try to reconstruct the UI hierarchy. Additionally, UIBert [186] aims to learn a general representation of UI designs and each of the UI elements to facilitate different downstream applications without the need to finetune the model. All these models can learn some low-level and high-level semantics about the UI during the training process, and the latent vectors have potential capabilities to enable UI search. Further evaluations are needed to confirm the capabilities.

7.2 Accessibility Enhancement

As the first work that tries to automatically fix the accessibility issues, our LabelDroid also has many spaces to improve. One direction is to incorporate the semantic information including the full screen, and the surrounding elements, to assist the label generation. Mehralian et al.[187] leverage app-level, activity-level, and icon-level information to empower the label generation model. Another line is to utilize the large-scale icon dataset in the wild to serve as a knowledge base to guide the label generation process. For example, Noun Project [1] contains many icons uploaded by designers. Each of the icons is well-annotated with different labels to depict all potential meanings, which has the potential to guide the accessibility icon generation process. One thing worthies to notice is that some labels in the training dataset may be wrong. In our work and Mehralian et al.'s work, we did not consider this issue. Future work could include a justification on the icon quality to better enhance the model's performance or correct the wrong one.

Apart from the accessibility issues of missing accessibility labels, other issues are also worth investigating. Liu et al. [78] spot the UI display issues in truncated texts which are caused by setting a larger font size, to improve usability. Latte [188] automatically detects the element unreachable problems for screen readers by simulating the usage of blind users, which adds a new tool for examining and enhancing the

[1] https://thenounproject.com/

accessibility of mobile applications. Chen et al. [189] extract the action sequence in the recorded usage video to try to guide the mobile application usage for people who are not familiar with mobile apps. All these works are important to improve the accessibility and usability of mobile apps.

7.3 UI Element Detection

GUI element detection plays an essential role in many downstream applications. Our work conducted the first systematically analysis of the unique characteristics of GUI and GUI elements and proposed the state-of-the-art GUI element detection tool (UIED) that leverages the strength of old-fashioned techniques and deep learning techniques. However, while our outperforms the best existing technique by 19.4% increase in F1 for non-text elements and 47.7% increase in F1 for all GUI elements, there are still many rooms to improve. One problem with our approach is that UIs with image background will bring some noise to our detected techniques. One potential mitigation method is to combine the noisy-resistance property of deep learning techniques with our techniques. For example, one could use deep learning techniques to provide initial localisation and further refine the detections using our technique. Another potential solution is to denoise the background like background removal tasks in photo editing. In addition, we found that the repetitive regions in a dense UI often have inconsistent detection results. Incorporating the surrounding information and understanding the relationship between components may have the potential to mitigate this issue. Some techniques like image inpainting may share similar ideas in recovering missing elements.

Chapter 8

Conclusions and Future Research

In this chapter, I summarize the research work that I have conducted in this thesis and discuss some future work.

8.1 Summary of Completed Work

With the advance of deep learning techniques and the availability of large-scale datasets, research and application have been accelerated in many sectors. DL techniques have demonstrated the ability to automatically learn the features from massive data, and thus reduce the need for manual feature engineering and rule-based methods. Although deep learning has made significant progress in the technology consumer domains, it has been much less explored from the technology producer perspective, i.e., those who develop the technology, for example, product designers and software developers.

In this thesis, I investigate deep learning techniques for software engineering domain, especially for enhancing the efficiency of mobile user interfaces development. I propose two tools and one fundamental work to assist UI designers and developers by leveraging UI-related big data with deep learning techniques.

First, to enhance the efficiency of the UI design prototyping process, I propose a wireframe-based UI design search engine to help designers and developers to understand the design space and software features by exposing them to a large-scale UI design dataset collected from real-world Android applications. I first collect a large-scale UI design dataset and then use a CNN-based image autoencoder to learn the latent vector of the UI design wireframe. I also develop a web-based search engine to implement the proposed method. This tool enables them to find design-wise similar UIs and quickly refine their designs. While I target the novice designers in this work, it can actually benefit designers with different levels of expertise [183; 190; 191].

Second, to assist the developers to implement some important yet easy-to-be-ignored accessibility features of user interfaces. I first conduct a motivational mining study to understand the severity of current accessibility issues of missing labels for screen readers. I then propose an efficient and useful tool, LabelDroid, based on the state-of-art image captioning technique to automatically generate labels for the image-based buttons. This method successfully invokes the community attention in maintaining the accessibility of mobile apps, many following research are published [35; 188; 78; 187].

Third, I systematically characterize the unique properties of UI elements and UIs, conduct the first large-scale empirical study on evaluating seven object detection models, and then propose our hybrid element detection method. Such methods can be integral in many applications, including the two techniques I propose in this thesis. In addition, this work can also enhance many other downstream applications, like UI code generation [73; 34; 185] and UI testing [77; 192].

The limitation and potential solutions of each work are dicussed in detail in Chapter 7

8.2 Future Work

The goal of my future research is to further provide more tools to assist developers from different aspects. Apart from improving my work to achieve better performance, there are still many applications that can be further investigated, such as multiple UI design search and generation, Accessibility Linting and Code Generation.

- **Multiple UI Design Search and Generation**: While there is many research on single UI design search, understanding the relationship among multiple UI designs is also important. Nowadays, mobile applications are comprised of many UIs and provide many functionalities, and it requires end-users to visit several UIs and perform multiple interactions to complete some tasks. If the path to the target function is hard to find, it will definitely bring a bad user experience to end-users, especially for end-users that have difficulty in using the mobile applications. Therefore, designing a set of UIs that have a straightforward connection and expose a clear path to the target function is important and necessary. Such task-level UI design understanding can also help the app testing process. For example, instead of randomly clicking the interacted UI elements on UIs to find a low-level bug, task-oriented UI testing is more practical and useful.

- **Simplifying User Interface**: Mobile applications are becoming more and more complex. For example, social media apps nowadays have not only the function of connecting with friends, but also enable users to do online shopping, find jobs, and play games. With so many features, UIs are becoming more sophisticated and this situation increases the cognitive burdens. A customized user interface is needed to suit one's preference as people will not use most of the features provided by apps. Moreover, the overwhelming features will also distract users' attention and users will easily forget what they want to do when some new stimulus appears. Automatically synthesizing a customized user interface based on the preference of the user will be beneficial for a better user experience.

- **Code Generation from UI screenshots**: As the development process of user interfaces will undergo several revisions, the developers need to re-implement user interfaces many times. For example, when UI testing teams find some usability or function-related issues, the designers may need to re-design some UIs and thus the developers need to revise the code according to the revisions. In addition, new features will gradually appear and old features will be replaced or removed as time goes. Automatically generating UI screenshots will definitely accelerate the development process as the repetitive and laborious work can be done automatically. There are already some attempts about generating the code from UI design [33; 34]. However, existing techniques mainly focus on single UI design code generation, without considering the connections between multiple UIs. For example, the developers will consider first creating some templates for a small part of UIs and then re-using these templates in multiple UIs. Extracting the templates will definitely reduce the workload of maintaining and revising the user interface.

Bibliography

1. Terry Winograd. From programming environments to environments for designing. *Communications of the ACM*, 38(6):65–74, 1995. (cited on pages 1 and 17)

2. Bernard J Jansen. The graphical user interface. *ACM SIGCHI Bulletin*, 30(2):22–26, 1998. (cited on pages 1 and 17)

3. A fish in your ear, 2018. (cited on pages 1 and 17)

4. Aliaksei Miniukovich and Antonella De Angeli. Pick me!: Getting noticed on google play. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 4622–4633. ACM, 2016. (cited on pages 1, 17, and 29)

5. Bardia Doosti, Tao Dong, Biplab Deka, and Jeffrey Nichols. A computational method for evaluating ui patterns. *arXiv preprint arXiv:1807.04191*, 2018. (cited on pages 1, 17, and 29)

6. Designing the ui of google translate, 2018. (cited on pages 1 and 17)

7. Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. Wireframe-based ui design search through image autoencoder. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(3):1–31, 2020. (cited on pages 1, 3, and 14)

8. Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang 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*, pages 665–676. ACM, 2018. (cited on pages 1, 3, 11, 12, 14, 19, 23, 30, 40, 52, and 71)

9. Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O Wobbrock. Interaction proxies for runtime repair and enhancement of mobile application accessibility. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 6024–6037. ACM, 2017. (cited on pages 1 and 13)

10. Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *42nd International Conference on Software Engineering (ICSE '20)*, page 13 pages, New York, NY, May 23-29, 2020 2020. ACM. (cited on pages 1, 3, 11, and 14)

11. Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. Automated reporting of gui design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 165–175, New York, NY, USA, 2018. Association for Computing Machinery. (cited on pages 1, 12, and 14)

12. Develop for android - material design, 2018. (cited on pages 2 and 17)

13. Human interface guidelines - design - apple developer, 2018. (cited on pages 2 and 17)

14. Appcelerator / idc 2015 mobile trends report: Data access is the new mobile problem. https://www.appcelerator.com/resource-center/research/2015-mobile-trends-report/, 2015. Accessed: 2018-04-25. (cited on pages 2 and 18)

15. Anil K Jain and Aditya Vailaya. Image retrieval using color and shape. *Pattern recognition*, 29(8):1233–1244, 1996. (cited on pages 2, 18, and 35)

16. David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999. (cited on pages 2, 18, and 35)

17. Steven P Reiss, Yun Miao, and Qi Xin. Seeking the user interface. *Automated Software Engineering*, 25(1):157–193, 2018. (cited on pages 2, 3, 11, 18, and 71)

18. Farnaz Behrang, Steven P Reiss, and Alessandro Orso. Guifetch: supporting app design and development through gui search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 236–246. ACM, 2018. (cited on pages 2, 11, 18, and 35)

19. Shuyu Zheng, Ziniu Hu, and Yun Ma. Faceoff: Assisting the manifestation design of web graphical user interface. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, page 774–777, New York, NY, USA, 2019. Association for Computing Machinery. (cited on pages 2, 11, 14, and 18)

20. Blindness and vision impairment. https://www.who.int/en/news-room/fact-sheets/detail/blindness-and-visual-impairment, 2018. (cited on pages 2 and 49)

21. Richard E Ladner. Design for user empowerment. *interactions*, 22(2):24–29, 2015. (cited on pages 3 and 49)

22. Android lint - android studio project site. http://tools.android.com/tips/lint, 2011. (cited on pages 3, 13, 50, and 51)

23. Accessibility scanner. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor, 2019. (cited on pages 3, 13, 50, and 51)

24. Espresso | android developers. https://developer.android.com/training/testing/espresso, 2019. (cited on pages 3, 14, 50, and 51)

25. Robolectric. http://robolectric.org/, 2019. (cited on pages 3, 14, 50, and 51)

26. Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192, 2009. (cited on pages 3, 14, 71, 74, and 77)

27. Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. Roscript: A visual script driven truly non-intrusive robotic testing system for touch screen applications. In *42nd International Conference on Software Engineering (ICSE '20)*, New York, NY, May 23-29, 2020 2020. ACM. (cited on pages 3, 14, 71, 74, and 77)

28. Thomas D White, Gordon Fraser, and Guy J Brown. Improving random gui testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 307–317, 2019. (cited on pages 3, 14, 15, 71, 72, 74, and 77)

29. Carlos Bernal-Cardenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *42nd International Conference on Software Engineering (ICSE '20)*, New York, NY, May 23-29, 2020 2020. ACM. (cited on pages 3, 14, and 71)

30. Morgan Dixon and James Fogarty. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1525–1534, 2010. (cited on pages 3, 14, 71, 74, and 77)

31. Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. Waken: reverse engineering usage information and interface structure from software videos. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 83–92, 2012. (cited on pages 3 and 71)

32. Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 845–854. ACM, 2017. (cited on pages 3, 11, 12, 14, 18, 23, 35, 71, 78, and 79)

33. Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 248–259. IEEE, 2015. (cited on pages 3, 11, 12, 14, 71, 72, 73, 74, 77, 78, 80, 90, 92, and 103)

34. Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, 46(2):196–221, 2020. (cited on pages 3, 11, 12, 14, 71, 72, 74, 77, 78, 94, 102, and 103)

35. Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021. (cited on pages 3, 98, and 102)

36. Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020. (cited on pages 3 and 98)

37. Screen reader survey. https://webaim.org/projects/screenreadersurvey7/, 2017. (cited on page 8)

38. Google talkback source code. https://github.com/google/talkback, 2019. (cited on pages 8, 13, and 49)

39. Voiceover. https://cloud.google.com/translate/docs/, 2019. (cited on pages 8, 13, and 49)

40. android.widget | android developers. https://developer.android.com/reference/android/widget/package-summary, 2018. (cited on page 8)

41. Imageview. https://developer.android.com/reference/android/widget/ImageView, 2019. (cited on page 9)

42. Image button. https://developer.android.com/reference/android/widget/ImageButton, 2019. (cited on page 9)

43. Five free ui kits for adobe xd created by top ux designers. https://theblog.adobe.com/five-top-ux-designers-five-ui-kits-adobe-xd-now-available-free/, 2018. Accessed: 2018-04-25. (cited on page 11)

44. 30+ great ui kits for ios engineers. https://medium.com/flawless-app-stories/30-great-ui-kits-for-ios-engineers-41b2732896b9, 2018. Accessed: 2018-04-25. (cited on page 11)

45. Essential free ui kits for mobile app designers. https://1stwebdesigner.com/ui-kits-for-mobile/, 2017. Accessed: 2018-04-25. (cited on page 11)

46. Free mobile ui kits. https://psddd.co/category/mobile-ui-kits/, 2018. Accessed: 2018-04-25. (cited on page 11)

47. Dribbble - discover the world's top designers & creative professionals, 2018. (cited on pages 11 and 18)

48. Ui movement - the best ui design inspiration, every day, 2018. (cited on pages 11 and 18)

49. Carlos Bernal-Cárdenas, Kevin Moran, Michele Tufano, Zichang Liu, Linyong Nan, Zhehan Shi, and Denys Poshyvanyk. Guigle: A gui search engine for android apps. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, page 71–74. IEEE Press, 2019. (cited on pages 11 and 18)

50. Daniel Ritchie, Ankita Arvind Kejriwal, and Scott R Klemmer. d. tour: Style-based exploration of design example galleries. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 165–174. ACM, 2011. (cited on pages 11 and 23)

51. Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. Gallery dc: Design search and knowledge discovery through auto-created gui component gallery. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–22, 2019. (cited on pages 11, 14, 15, 72, 74, and 77)

52. Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R Klemmer, and Jerry O Talton. Webzeitgeist: design mining the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3083–3092. ACM, 2013. (cited on pages 11 and 23)

53. Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Seenomaly: Vision-based linting of gui animation effects against design-don't guidelines. In *42nd International Conference on Software Engineering (ICSE '20)*, page 12 pages, New York, NY, May 23-29, 2020 2020. ACM. (cited on pages 11, 14, and 71)

54. Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. Storydroid: Automated generation of storyboard for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 596–607. IEEE, 2019. (cited on pages 11 and 14)

55. Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. Gui-squatting attack: Automated generation of android phishing apps. *IEEE Transactions on Dependable and Secure Computing*, 2019. (cited on pages 11 and 14)

56. Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 3. ACM, 2018. (cited on page 12)

57. Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. Detecting and summarizing gui changes in evolving mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 543–553, New York, NY, USA, 2018. Association for Computing Machinery. (cited on page 12)

58. Android accessibility guideline. https://developer.android.com/guide/topics/ui/accessibility/apps, 2019. (cited on pages 12, 50, 51, and 64)

59. ios accessibiliyu guideline. https://developer.apple.com/accessibility/ios/, 2019. (cited on page 12)

60. World wide web consortium accessibility. https://www.w3.org/standards/webdesign/accessibility, 2019. (cited on page 12)

61. Shunguo Yan and PG Ramachandran. The current status of accessibility in mobile apps. *ACM Transactions on Accessible Computing (TACCESS)*, 12(1):3, 2019. (cited on page 12)

62. Kyudong Park, Taedong Goh, and Hyo-Jeong So. Toward accessible mobile application design: developing mobile application accessibility guidelines for people with visual impairment. In *Proceedings of HCI Korea*, pages 31–38. Hanbit Media, Inc., 2014. (cited on pages 12 and 13)

63. Fahui Wang. Measurement, optimization, and impact of health care accessibility: a methodological review. *Annals of the Association of American Geographers*, 102(5):1104–1112, 2012. (cited on page 12)

64. Higinio Mora, Virgilio Gilart-Iglesias, Raquel Pérez-del Hoyo, and María Andújar-Montoya. A comprehensive system for monitoring urban accessibility in smart cities. *Sensors*, 17(8):1834, 2017. (cited on page 12)

65. Bridgett A King and Norman E Youngblood. E-government in alabama: An analysis of county voting and election website content, usability, accessibility, and mobile readiness. *Government Information Quarterly*, 33(4):715–726, 2016. (cited on page 12)

66. Shaun K Kane, Chandrika Jayant, Jacob O Wobbrock, and Richard E Ladner. Freedom to roam: a study of mobile device adoption and accessibility for people with visual and motor disabilities. In *Proceedings of the 11th international ACM SIGACCESS conference on Computers and accessibility*, pages 115–122. ACM, 2009. (cited on page 13)

67. Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. Examining image-based button labeling for accessibility in android apps through large-scale analysis. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 119–130. ACM, 2018. (cited on page 13)

68. Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon Fraser. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126. IEEE, 2018. (cited on page 13)

69. Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement. In *The 31st Annual ACM Symposium on User Interface Software and Technology*, pages 609–621. ACM, 2018. (cited on page 13)

70. Earl-grey. https://github.com/google/EarlGrey, 2019. (cited on page 14)

71. Kif. https://github.com/kif-framework/KIF, 2019. (cited on page 14)

72. Android's accessibility testing framework. https://github.com/google/Accessibility-Test-Framework-for-Android, 2019. (cited on page 14)

73. Pavol Bielik, Marc Fischer, and Martin Vechev. Robust relational layout synthesis from examples for android. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018. (cited on pages 14 and 102)

74. Forrest Huang, John F Canny, and Jeffrey Nichols. Swire: Sketch-based user interface retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–10, 2019. (cited on page 14)

75. Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. From lost to found: Discover missing ui design semantics through recovering missing tags. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW), 2020. (cited on page 14)

76. Amanda Swearngin and Yang Li. Modeling mobile interface tappability using crowdsourcing and deep learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2019. (cited on pages 14 and 94)

77. Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1070–1073. IEEE, 2019. (cited on pages 14 and 102)

78. Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl eyes: Spotting ui display issues via visual understanding. In *Proceedings of the 35th International Conference on Automated Software Engineering*, 2020. (cited on pages 14, 98, and 102)

79. Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng 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*, pages 2421–2436, 2019. (cited on page 14)

80. Chen Yongxin, Zhang Tonghui, and Chen Jie. Ui2code: How to fine-tune background and foreground analysis, May 2019. (cited on pages 14, 72, 73, 74, 77, 78, 80, 90, 92, and 95)

81. Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, and Bo Zhou. scvripper: video scraping tool for modeling developers' behavior using interaction data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 673–676. IEEE, 2015. (cited on pages 14, 74, and 77)

82. J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, Nov 1986. (cited on pages 14, 76, 77, 80, and 92)

83. Satoshi Suzuki and KeiichiA be. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32 – 46, 1985. (cited on pages 14, 76, 77, 91, and 92)

84. Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015. (cited on pages 15, 55, 72, 74, 77, 78, 80, and 90)

85. Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018. (cited on pages 15, 72, 74, 77, 78, 80, and 81)

86. Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. Centernet: Keypoint triplets for object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 6569–6578, 2019. (cited on pages 15, 72, 74, 77, 78, 80, and 90)

87. Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017. (cited on pages 15, 74, and 90)

88. Jason Hong. Matters of design. *Commun. ACM*, 54(2):10–11, February 2011. (cited on pages 18 and 20)

89. Why is good ui design so hard for some developers?, 2018. (cited on page 18)

90. Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994. (cited on page 18)

91. Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 762–771. IEEE Press, 2013. (cited on page 18)

92. Xin Peng, Zhenchang Xing, Xi Tan, Yijun Yu, and Wenyun Zhao. Iterative context-aware feature location (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 900–903. ACM, 2011. (cited on page 18)

93. Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120. ACM, 2011. (cited on page 18)

94. Guibin Chen, Chunyang Chen, Zhenchang Xing, and Bowen Xu. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 744–755. IEEE, 2016. (cited on pages 19 and 55)

95. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. (cited on pages 19, 26, and 55)

96. Lingfeng Bao, Zhenchang Xing, Xinyu Wang, and Bo Zhou. Tracking and analyzing cross-cutting activities in developers' daily work (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 277–282. IEEE, 2015. (cited on page 23)

97. Android documentation - user interface & navigation, 2019. (cited on page 24)

98. Adobe xd, 2020. (cited on pages 25, 26, and 31)

99. Fluid ui, 2020. (cited on pages 25, 26, and 31)

100. Balsamiq mockups, 2020. (cited on pages 25, 26, and 31)

101. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015. (cited on page 26)

102. Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015. (cited on page 26)

103. Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *J. Mach. Learn. Res.*, 11:3371–3408, December 2010. (cited on page 26)

104. Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014. (cited on page 26)

105. Naila Murray and Florent Perronnin. Generalized max pooling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2473–2480, 2014. (cited on page 27)

106. Robert Keys. Cubic convolution interpolation for digital image processing. *IEEE transactions on acoustics, speech, and signal processing*, 29(6):1153–1160, 1981. (cited on page 27)

107. James O Berger. *Statistical decision theory and Bayesian analysis*. Springer Science & Business Media, 2013. (cited on page 29)

108. Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010. (cited on page 29)

109. Thought Leadership White Paper. IBM Software. Native, web or hybrid mobile-app development. http://cdn.computerworld.com.au/whitepaper/371126/native-web-or-hybrid-mobile-app-development/download/?type=other&arg=0&location=featured_list, 2012. (cited on page 29)

110. Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. (cited on page 30)

111. Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 182–191. IEEE, 2013. (cited on page 31)

112. Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Long Xiong Ong. Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding. *IEEE Transactions on Software Engineering*, 2019. (cited on pages 31 and 55)

113. Jing Li, Zhenchang Xing, and Aixin Sun. Linklive: discovering web learning resources for developers from q&a discussions. *World Wide Web*, 22(4):1699–1725, 2019. (cited on page 31)

114. Bowen Xu, Zhenchang Xing, Xin Xia, David Lo, and Shanping Li. Domain-specific cross-language relevant question retrieval. *Empirical Software Engineering*, 23(2):1084–1122, 2018. (cited on page 31)

115. Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. Answerbot: automated generation of answer summary to developersź technical questions. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 706–716. IEEE Press, 2017. (cited on page 31)

116. Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017. (cited on page 31)

117. Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings*

of the IEEE conference on computer vision and pattern recognition, pages 113–123, 2019. (cited on page 31)

118. Chunyang Chen, Xi Chen, Jiamou Sun, Zhenchang Xing, and Guoqiang Li. Data-driven proactive policy assurance of post quality in community q&a sites. *Proceedings of the ACM on human-computer interaction*, 2(CSCW):1–22, 2018. (cited on pages 34 and 55)

119. Ju Han and Kai-Kuang Ma. Fuzzy color histogram and its use in color image retrieval. *IEEE Transactions on image Processing*, 11(8):944–952, 2002. (cited on page 35)

120. James Hafner, Harpreet S. Sawhney, William Equitz, Myron Flickner, and Wayne Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE transactions on pattern analysis and machine intelligence*, 17(7):729–736, 1995. (cited on page 35)

121. Yan Ke, Rahul Sukthankar, Larry Huston, Yan Ke, and Rahul Sukthankar. Efficient near-duplicate detection and sub-image retrieval. In *Acm Multimedia*, volume 4, page 5. Citeseer, 2004. (cited on page 35)

122. Zhong Wu, Qifa Ke, Michael Isard, and Jian Sun. Bundling features for large scale partial-duplicate web image search. In *Computer vision and pattern recognition, 2009. cvpr 2009. ieee conference on*, pages 25–32. IEEE, 2009. (cited on page 35)

123. Matthew Brown and David G Lowe. Automatic panoramic image stitching using invariant features. *International journal of computer vision*, 74(1):59–73, 2007. (cited on page 35)

124. Richard M Karp, Umesh V Vazirani, and Vijay V Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 352–358. ACM, 1990. (cited on page 35)

125. Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960. (cited on page 41)

126. Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971. (cited on page 41)

127. Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016. (cited on page 46)

128. Zhou , Alan C Bovik, Hamid R Sheikh, Eero P Simoncelli, et al. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004. (cited on page 46)

129. Nikita Prabhu and R Venkatesh Babu. Attribute-graph: A graph based approach to image ranking. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1071–1079, 2015. (cited on page 46)

130. Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012. (cited on pages 46 and 47)

131. Google play store. https://play.google.com, 2019. (cited on pages 49 and 52)

132. Apple app store. https://www.apple.com/au/ios/app-store/, 2019. (cited on page 49)

133. Ch Spearman. The proof and measurement of association between two things. *International journal of epidemiology*, 39(5):1137–1150, 2010. (cited on page 54)

134. Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995. (cited on page 55)

135. Chunyang Chen, Zhenchang Xing, and Yang Liu. By the community & for the community: a deep learning approach to assist collaborative editing in q&a sites. *Proceedings of the ACM on Human-Computer Interaction*, 1(CSCW):1–21, 2017. (cited on page 55)

136. Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. A neural model for method name generation from functional description. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 414–421. IEEE, 2019. (cited on page 55)

137. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. (cited on page 55)

138. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. (cited on pages 55, 57, 59, 73, 81, and 90)

139. Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. Actionnet: vision-based workflow action recognition from programming screencasts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 350–361. IEEE, 2019. (cited on page 55)

140. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. (cited on pages 56 and 61)

141. Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. (cited on page 57)

142. Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951. (cited on pages 58 and 60)

143. Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. (cited on page 59)

144. Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. (cited on page 60)

145. Pytorch. https://pytorch.org/s, 2019. (cited on page 60)

146. Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002. (cited on page 60)

147. Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005. (cited on pages 60 and 61)

148. Chin-Yew Lin and Eduard Hovy. Automatic evaluation of summaries using n-gram co-occurrence statistics. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 150–157, 2003. (cited on pages 60 and 61)

149. Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4566–4575, 2015. (cited on pages 60 and 61)

150. Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. Piscataway, NJ, 2003. (cited on page 61)

151. Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco captions: Data collection and evaluation server. *arXiv preprint arXiv:1504.00325*, 2015. (cited on page 61)

152. Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015. (cited on page 61)

153. Jyoti Aneja, Aditya Deshpande, and Alexander G Schwing. Convolutional image captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5561–5570, 2018. (cited on page 61)

154. Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947. (cited on page 62)

155. Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300, 1995. (cited on pages 62 and 66)

156. Talkback guideline. https://support.google.com/accessibility/android/answer/6283677?hl=en, 2019. (cited on page 64)

157. Isao Goto, Ka-Po Chow, Bin Lu, Eiichiro Sumita, and Benjamin K Tsou. Overview of the patent machine translation task at the ntcir-10 workshop. In *NTCIR*, 2013. (cited on page 65)

158. John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996. (cited on page 65)

159. Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 574–584. IEEE, 2015. (cited on page 65)

160. Frank Wilcoxon. Individual comparisons by ranking methods. In *Breakthroughs in statistics*, pages 196–202. Springer, 1992. (cited on page 66)

161. Feng Lin, Chen Song, Xiaowei Xu, Lora Cavuoto, and Wenyao Xu. Sensing from the bottom: Smart insole enabled patient handling activity recognition through manifold learning. In *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 254–263. IEEE, 2016. (cited on page 71)

162. Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F Cohen. Pause-and-play: automatically linking screencast video tutorials with applications. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 135–144, 2011. (cited on page 71)

163. Lingfeng Bao, Deheng Ye, Zhenchang Xing, Xin Xia, and Xinyu Wang. Activityspace: a remembrance framework to support interapplication information needs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 864–869. IEEE, 2015. (cited on page 71)

164. Karl Bridge and Michael Satran. Windows accessibility api overview, May 2018. (cited on page 71)

165. Google. Build more accessible apps, 2020. (cited on page 71)

166. Microsoft. Introducing spy++, November 2016. (cited on page 71)

167. Google. Ui automator, December 2019. (cited on page 71)

168. Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Andrew J Ko. Rewire: Interface design assistance from examples. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 504. ACM, 2018. (cited on page 71)

169. Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5551–5560, 2017. (cited on pages 73, 78, 81, 90, and 93)

170. Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014. (cited on pages 75 and 76)

171. Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010. (cited on page 76)

172. Ray Smith. An overview of the tesseract ocr engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 2, pages 629–633. IEEE, 2007. (cited on pages 78 and 80)

173. Mohian Soumik. pix2app, 2019. (cited on page 80)

174. Shane Torbert. *Applied computer science*. Springer, 2016. (cited on pages 80 and 91)

175. Jing Yang, Qingshan Liu, and Kaihua Zhang. Stacked hourglass network for robust facial landmark localisation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 79–87, 2017. (cited on page 81)

176. Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, Sep 2009. (cited on page 90)

177. Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244–256, 1972. (cited on page 91)

178. Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Addison-Wesley, 1993. (cited on page 92)

179. Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Dorling Kindersley, 2014. (cited on page 92)

180. H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, 1988. (cited on page 92)

181. Alasdair Tran, Alexander Mathews, and Lexing Xie. Transform and tell: Entity-aware news image captioning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13035–13045, 2020. (cited on page 97)

182. Dipu Manandhar, Dan Ruta, and John Collomosse. Learning structural similarity of user interface layouts using graph networks. In *European Conference on Computer Vision*, pages 730–746. Springer, 2020. (cited on page 97)

183. Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. Screen2vec: Semantic embedding of gui screens and gui components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021. (cited on pages 97 and 101)

184. Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. Screen2words: Automatic mobile ui summarization with multimodal learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 498–510, 2021. (cited on page 98)

185. Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. Screen parsing: Towards reverse engineering of ui models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 470–483, 2021. (cited on pages 98 and 102)

186. Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, et al. Uibert: Learning generic multimodal representations for ui understanding. *arXiv preprint arXiv:2107.13731*, 2021. (cited on page 98)

187. Forough Mehralian, Navid Salehnamadi, and Sam Malek. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 107–118, 2021. (cited on pages 98 and 102)

188. Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2021. (cited on pages 98 and 102)

189. Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. Extracting replayable interactions from videos of mobile app usage. *arXiv preprint arXiv:2207.04165*, 2022. (cited on page 99)

190. Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. Vins: Visual search for mobile user interface design. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2021. (cited on page 101)

191. Chunggi Lee, Sanghoon Kim, Dongyun Han, Hongjun Yang, Young-Woo Park, Bum Chul Kwon, and Sungahn Ko. Guicomp: A gui design assistant with real-time, multi-faceted feedback. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020. (cited on page 101)

192. Shengcheng Yu, Chunrong Fang, Yang Feng, Wenyuan Zhao, and Zhenyu Chen. Lirat: Layout and image recognition driving automated mobile testing of cross-platform. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1066–1069. IEEE, 2019. (cited on page 102)