Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning

Jieshan Chen
Jieshan.Chen@anu.edu.au
Australian National University
Australia

Chunyang Chen*
Chunyang.Chen@monash.edu
Monash University
Australia

Zhenchang Xing†
Zhenchang.Xing@anu.edu.au
Australian National University
Australia

Xiwei Xu
Xiwei.Xu@data61.csiro.au
Data61, CSIRO
Australia

Liming Zhu†‡
Liming.Zhu@data61.csiro.au
Australian National University
Australia

Guoqiang Li*
Li.G@sjtu.edu.cn
Shanghai Jiao Tong University
China

Jinshui Wang*
ymkscom@gmail.com
Fujian University of Technology
China

ABSTRACT

According to the World Health Organization(WHO), it is estimated that approximately 1.3 billion people live with some forms of vision impairment globally, of whom 36 million are blind. Due to their disability, engaging these minority into the society is a challenging problem. The recent rise of smart mobile phones provides a new solution by enabling blind users' convenient access to the information and service for understanding the world. Users with vision impairment can adopt the screen reader embedded in the mobile operating systems to read the content of each screen within the app, and use gestures to interact with the phone. However, the prerequisite of using screen readers is that developers have to add natural-language labels to the image-based components when they are developing the app. Unfortunately, more than 77% apps have issues of missing labels, according to our analysis of 10,408 Android apps. 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 description 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.

CCS CONCEPTS

• **Human-centered computing** → **Accessibility systems and tools**; *Empirical studies in accessibility*; • **Software and its engineering** → **Software usability**.

KEYWORDS

Accessibility, neural networks, user interface, image-based buttons, content description

ACM Reference Format:
Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380327

1 INTRODUCTION

Given millions of mobile apps in Google Play [11] and App store [8], 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 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) [4], 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

*Corresponding author.
†Also with Data61, CSIRO.
‡Also with University of New South Wales.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380327

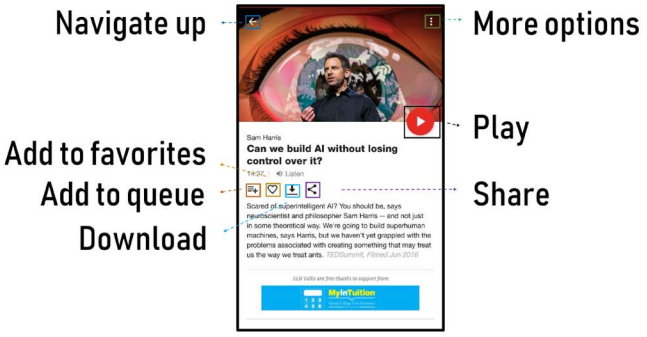

Figure 1: Example of UI components and labels.

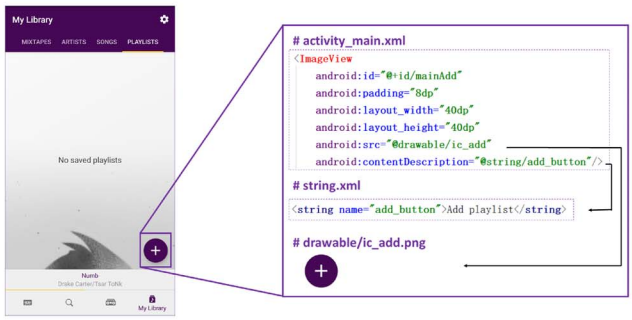

Figure 2: Source code for setting up labels for "add playlist" button (which is indeed a *clickable ImageView*).

represent their eyes. Ensuring full access to the wealth of information and services provided by mobile apps is a matter of social justice [54].

Fortunately, the mobile platforms have begun to support app accessibility by screen readers (e.g., TalkBack in Android [12] and VoiceOver in IOS [20]) 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 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 to "*provide content descriptions for UI components that do not have visible text*" [6]. 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 [6]. However, according to our empirical study in Section 3, 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 [1], Accessibility Scanner [5], and other accessibility testing frameworks [10, 18]. 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 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 decoder for predicting the labels based on the large-scale dataset. The experiments show that our LabelDroid can achieve 65% exact match and 0.697 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[2] for enabling other researchers' future research.

2 ANDROID ACCESSIBILITY BACKGROUND

2.1 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`

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

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

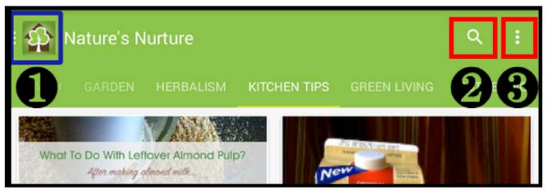

Figure 3: Examples of image-based buttons. ① `clickable ImageView`; ②③ `ImageButton`.

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 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.

2.2 Screen Reader

According to the screen reader user survey by WebAIM in 2017 [2], 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 [12] and VoiceOver in IOS [20]) 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. 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. 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.

2.3 Android Classes of Image-Based Buttons

There are many different types of UI components [3] 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 3 shows two kinds of image-based buttons.

Table 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%)

2.3.1 Clickable Images. Images can be rendered in an app using the Android API class `android.widget.ImageView`[14]. If the clickable property is set to true, the image functions as a button (① in Figure 3). 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.

2.3.2 Image Button. Image Buttons are implemented by the Android API class `android.widget.ImageButton` [13]. 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 3).

3 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.

3.1 Data Collection

To investigate how well the apps support the users with vision impairment, we randomly crawl 19,127 apps from Google Play [11], belonging to 25 categories with the installation number ranging from 1K to 100M.

We adopt the app explorer [32] 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 buttons including clickable images and image buttons, which forms the dataset we analyse in this study.

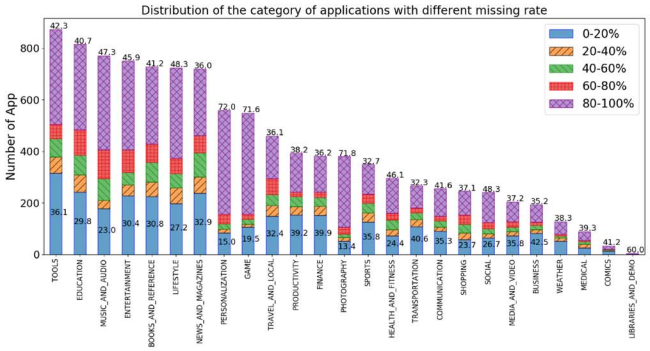

Figure 4: The distribution of the category of applications with different rate of image-based buttons missing content description

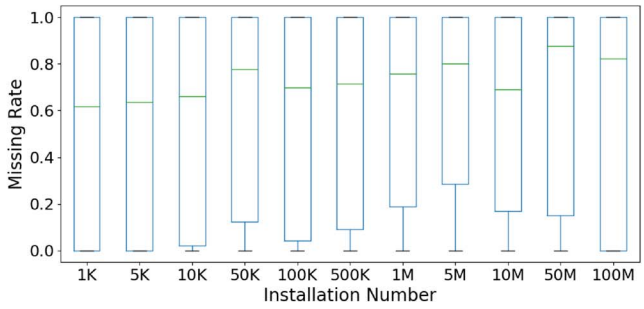

Figure 5: Box-plot for missing rate distribution of all apps with different installation numbers.

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

Table 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 4, 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 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. 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 [67] 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.

4 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 [55], 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 [33, 34, 44], we adopt the Transformer model [68] in this work. The overview of our approach can be seen in Figure 6.

4.1 Visual Feature Extraction

To extract the visual features from the input button image, we adopt the convolutional neural network [46] which is widely used in software engineering domain [30, 35, 79]. CNN-based model can automatically learn latent features from a large image database, which has outperformed hand-crafted features in many computer vision tasks [52, 65]. 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

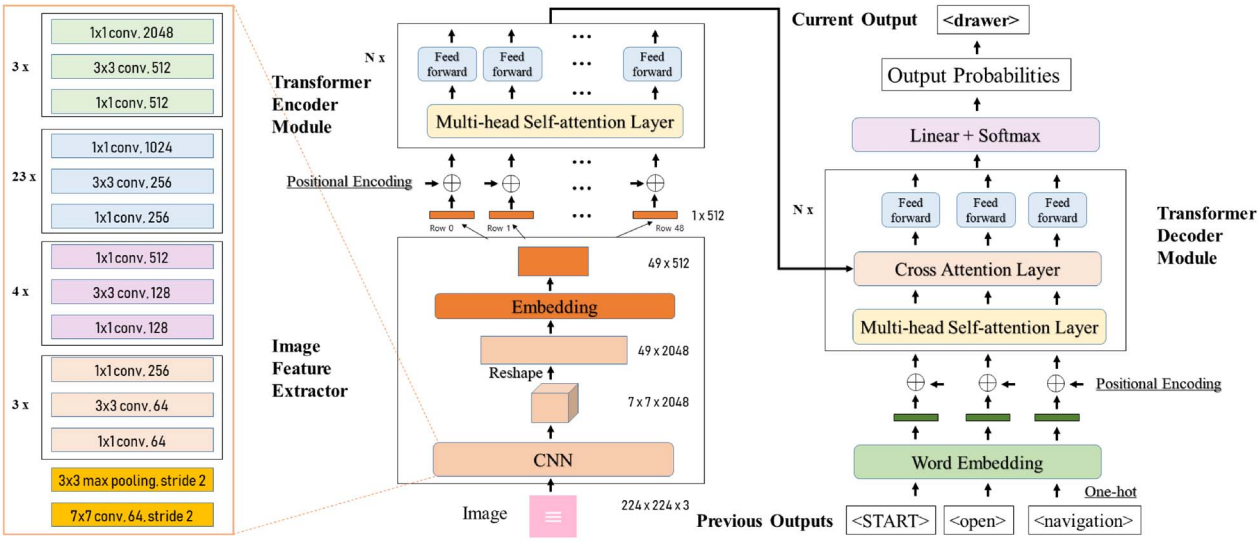

Figure 6: Overview of our approach

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.

4.2 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 [48], 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:

$$Self\_Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

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:

$$Feed\_forward(Z) = W_2^e \times (W_1^e \times Z + b_1^e) + b_2^e \quad (2)$$

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 [46] and layer normalization [23]. 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.

4.3 Content Description Generation

As mentioned in Section 4.2, 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:

$$Cross\_Attention(Q^e, K^e, V^d) = softmax(\frac{Q^e (K^e)^T}{\sqrt{d_k}}) V^d \quad (3)$$

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 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 [53] (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.

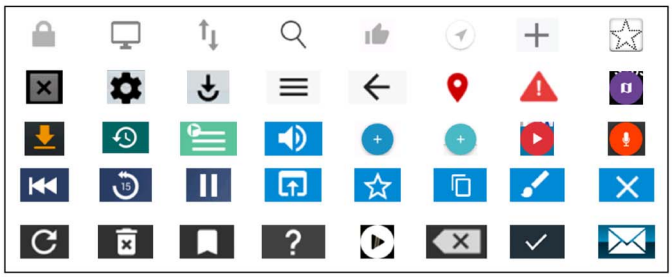

Figure 7: Example of our dataset

5 IMPLEMENTATION

To implement our model, we further extract the data analysed in Section 3 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.

5.1 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 with a different background within the same app. For example, for the "back" button at the top-left position in Figure 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. We manually observe 500 randomly selected image-based buttons from Section 3, 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 $< start >$, $< end >$

Table 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

tokens to the start and the end of the sequence. We also replace the low-frequency words (less than five) 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 3 as the apps with no or uninformative labels are removed. We split cleaned dataset into training, validation[3] and testing set. For each app category, we randomly select 80% apps for training, 10% for validation and the rest 10% for testing. Table 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.

5.2 Model Implementation

We use ResNet-101 architecture [46] pretrained on MS COCO dataset [57] as our CNN module. As you can see in the leftmost of Figure 6, it consists of a convolution 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 $N = 3$, $d_{embed} = 512$, $d_k = d_v = d_{model} = 64$, $d_{ff} = 2048$, $h = 8$ and the vocabulary size as 633. We train the CNN and the encoder-decoder model in an end-to-end manner using KL divergence loss [53]. We use Adam optimizer [51] 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 [17] on a machine with Intel i7-7800X CPU, 64G RAM and NVIDIA GeForce GTX 1080 Ti GPU.

[3]for tuning the hyperparameters and preventing the overfitting

6 EVALUATION

We evaluate our LabelDroid in three aspects, i.e., accuracy with automated testing, generality and usefulness with user study.

6.1 Evaluation Metric

To evaluate the performance of our model, we adopt five widely-used evaluation metrics including exact match, BLEU [62], METEOR [26], ROUGE [56], CIDEr [69] 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 [26] (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) [56] 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) [69] uses term frequency inverse document frequency (tf-idf) [64] 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 [39].

6.2 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 [48, 70]. The second baseline also adopt the encoder-decoder framework. Although it adopts the CNN model as

Table 3: Results of accuracy evaluation

Method	Exact match	BLEU@1	BLEU@2	BLEU@3	BLEU@4	METEOR	ROUGE-L	CIDEr
CNN+LSTM	62.5%	0.669	0.648	0.545	0.461	0.420	0.667	0.317
CNN+CNN	62.5%	0.667	0.645	0.555	0.503	0.421	0.669	0.318
LabelDroid	**65.0%**	**0.689**	**0.663**	**0.566**	**0.513**	**0.444**	**0.697**	**0.333**

the encoder, but uses another CNN model for generating the output [22] 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.

6.3 Accuracy Evaluation

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.

6.3.1 Overall Performance. Table 3 shows the overall performance of all methods. The performance of two baselines are very similar and they all achieve 62.5% exactly match rate. But CNN+CNN model is slightly higher in other metrics. In contrast with baselines, the generated labels from our LabelDroid for 65% image-based buttons exactly match the ground truth. And the average BLEU@1, BLEU@2, BLEU@3, BLEU@4, METEOR, ROUGE-L and CIDEr of our method are 0.689, 0.663, 0.566, 0.513, 0.444, 0.697, 0.333. 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 [59] between these three models among all testing metrics. Since we have three inferential statistical tests, we apply the Benjamini & Hochberg (BH) method [27] to correct p-values. Results show the improvement of our model is significant in all comparisons (p-value<0.01)[4].

To show the generalization of our model, we also calculate the performance of our model in different app categories as seen in Figure 8. We find that our model is not sensitive to the app category i.e., steady performance across different app categories. In addition, Figure 8 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.

6.3.2 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 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

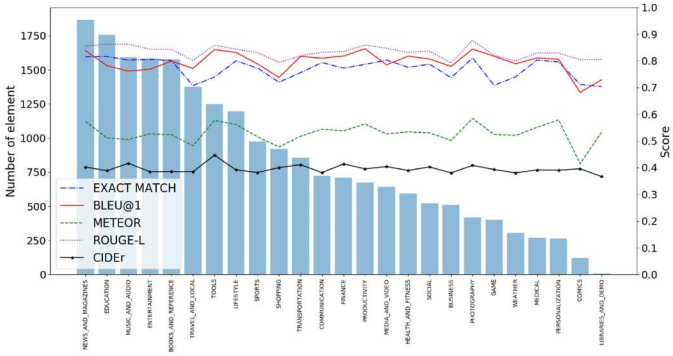

Figure 8: Performance distribution of different app category

"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.

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.

6.3.3 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 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 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 convey the same meaning which can

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

Table 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

Table 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

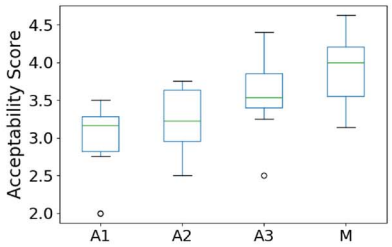

Figure 9: Distribution of app acceptability scores by human annotators (A1, A2, A3) and the model (M).

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.

6.4 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.

6.4.1 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 feed 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 [6, 19] and we demo them the example labels for some randomly selected image-based buttons (not in our dataset). During the experiment, they are shown the target image-based buttons highlighted in the whole UI (similar to Figure 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 [45]. Given one predicted content description for the button, the human evaluator will assign 5-point Likert scale [28, 61]) 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.

6.4.2 Results. Table 6[5] 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

[5]Detailed results are listed in https://github.com/chenjshnn/LabelDroid

Table 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)

Table 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

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 9 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 [74] 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)[6].

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 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 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.

7 RELATED WORK

Mobile devices are ubiquitous, and mobile apps are widely used for different tasks in people's daily life. Consequently, there are many research works for ensuring the quality of mobile apps [29, 43, 47]. Most of them are investigating the apps' functional and non-functional properties like compatibility [73], performance [58, 81], energy-efficiency [24, 25], GUI design [31, 36], GUI animation linting [80], localization [72] and privacy and security [37, 38, 40,

[6]The p-values are adjusted by Benjamin & Hochberg method [27]. All detailed p-values are listed in https://github.com/chenjshnn/LabelDroid

42, 76]. However, few of them are studying the accessibility issues, especially for users with vision impairment which is focused in our work.

7.1 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 [6, 15] which include not only the accessibility design principles, but also the documents for using assistive technologies embedding in the operating system [15], and testing tools or suits for ensuring the app accessibility. The World Wide Web Consortium (W3C) has released their web accessibility guideline long time ago [21] And now they are working towards adapting their web accessibility guideline [21] 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.

7.2 App Accessibility Studies for Blind Users

Many works in Human-Computer Interaction area have explored the accessibility issues of small-scale mobile apps [63, 75] in different categories such as in health [71], smart cities [60] and government engagement [50]. 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 [63] 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 [49] carry out a study of mobile device adoption and accessibility for people with visual and motor disabilities. Ross et al [66] 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, our study includes not only the largest-scale analysis of image-based button labeling 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 [41] develop an automated test generation model to dynamically test the mobile apps. Zhang et al [78] 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 [77] without referring to the source code. Although these works can also help ensure the quality of mobile accessibility, they still need much effort from developers. Instead, the model proposed in our work can automatically recommend the label for image-based components and developers can directly use it or modify it for their own apps.

7.3 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 [12] for Android, VoiceOver [20] 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 [5] scans the specified screen and provides suggestions to improve the accessibility of your app including content labels, clickable items, color contrast, etc. 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.

Second, developers can also automate accessibility tasks by resorting testing frameworks like Android Lint, Espresso and Robolectric, etc. The Android Lint [1] 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 [10] and Robolectric [18] which can also check accessibility issues dynamically during the testing execution. And there are counterparts for IOS apps like Earl-Grey [9] and KIF [16]. 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 [7].

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 our work is to develop a proactive content labeling model which can complement the reactive testing mechanism.

8 CONCLUSION AND FUTURE WORK

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.

We hope that this work can invoking the 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.

REFERENCES

[1] 2011. Android Lint - Android Studio Project Site. http://tools.android.com/tips/lint.
[2] 2017. Screen Reader Survey. https://webaim.org/projects/screenreadersurvey7/.

[3] 2018. android.widget | Android Developers. https://developer.android.com/reference/android/widget/package-summary.

[4] 2018. Blindness and vision impairment. https://www.who.int/en/news-room/fact-sheets/detail/blindness-and-visual-impairment.

[5] 2019. Accessibility Scanner. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor.

[6] 2019. Android Accessibility Guideline. https://developer.android.com/guide/topics/ui/accessibility/apps.

[7] 2019. Android's Accessibility Testing Framework. https://github.com/google/Accessibility-Test-Framework-for-Android.

[8] 2019. Apple App Store. https://www.apple.com/au/ios/app-store/.

[9] 2019. Earl-Grey. https://github.com/google/EarlGrey.

[10] 2019. Espresso | Android Developers. https://developer.android.com/training/testing/espresso.

[11] 2019. Google Play Store. https://play.google.com.

[12] 2019. Google TalkBack source code. https://github.com/google/talkback.

[13] 2019. Image Button. https://developer.android.com/reference/android/widget/ImageButton.

[14] 2019. ImageView. https://developer.android.com/reference/android/widget/ImageView.

[15] 2019. iOS Accessibiliyu Guideline. https://developer.apple.com/accessibility/ios/.

[16] 2019. KIF. https://github.com/kif-framework/KIF.

[17] 2019. PyTorch. https://pytorch.org/s.

[18] 2019. Robolectric. http://robolectric.org/.

[19] 2019. Talkback Guideline. https://support.google.com/accessibility/android/answer/6283677?hl=en.

[20] 2019. VoiceOver. https://cloud.google.com/translate/docs/.

[21] 2019. World Wide Web Consortium Accessibility. https://www.w3.org/standards/webdesign/accessibility.

[22] Jyoti Aneja, Aditya Deshpande, and Alexander G Schwing. 2018. Convolutional image captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5561–5570.

[23] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[24] Abhijeet Banerjee, Hai-Feng Guo, and Abhik Roychoudhury. 2016. Debugging energy-efficiency related field failures in mobile apps. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 127–138.

[25] Abhijeet Banerjee and Abhik Roychoudhury. 2016. Automated re-factoring of android apps to enhance energy-efficiency. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 139–150.

[26] Satanjeev Banerjee and Alon Lavie. 2005. 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*. 65–72.

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

[28] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[29] Margaret Butler. 2010. Android: Changing the mobile landscape. *IEEE Pervasive Computing* 10, 1 (2010), 4–7.

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

[31] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery DC: Design Search and Knowledge Discovery through Auto-created GUI Component Gallery. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–22.

[32] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. 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*. ACM, 665–676.

[33] Chunyang Chen, Zhenchang Xing, and Yang Liu. 2017. 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 (2017), 1–21.

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

[35] Guibin Chen, Chunyang Chen, Zhenchang Xing, and Bowen Xu. 2016. 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)*. IEEE, 744–755.

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

[37] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. 2019. GUI-Squatting Attack: Automated Generation of Android Phishing Apps. *IEEE Transactions on Dependable and Secure Computing* (2019).

[38] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security* 73 (2018), 326–344.

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

[40] Tobias Dehling, Fangjian Gao, Stephan Schneider, and Ali Sunyaev. 2015. Exploring the far side of mobile health: information security and privacy of mobile health apps on iOS and Android. *JMIR mHealth and uHealth* 3, 1 (2015), e8.

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

[42] Ruitao Feng, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, and Shang-Wei Lin. 2019. MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 61–70.

[43] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. 2013. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1276–1284.

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

[45] Isao Goto, Ka-Po Chow, Bin Lu, Eiichiro Sumita, and Benjamin K Tsou. 2013. Overview of the Patent Machine Translation Task at the NTCIR-10 Workshop.. In *NTCIR*.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[47] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the software quality of android applications along their evolution (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 236–247.

[48] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[49] Shaun K Kane, Chandrika Jayant, Jacob O Wobbrock, and Richard E Ladner. 2009. 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*. ACM, 115–122.

[50] Bridgett A King and Norman E Youngblood. 2016. E-government in Alabama: An analysis of county voting and election website content, usability, accessibility, and mobile readiness. *Government Information Quarterly* 33, 4 (2016), 715–726.

[51] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[53] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics* 22, 1 (1951), 79–86.

[54] Richard E Ladner. 2015. Design for user empowerment. *interactions* 22, 2 (2015), 24–29.

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

[56] Chin-Yew Lin and Eduard Hovy. 2003. 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*. 150–157.

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

[58] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How developers detect and fix performance bottlenecks in Android apps. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 352–361.

[59] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[60] Higinio Mora, Virgilio Gilart-Iglesias, Raquel Pérez-del Hoyo, and María Andújar-Montoya. 2017. A comprehensive system for monitoring urban accessibility in smart cities. *Sensors* 17, 8 (2017), 1834.

[61] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. 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*. IEEE, 574–584.

[62] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[63] Kyudong Park, Taedong Goh, and Hyo-Jeong So. 2014. Toward accessible mobile application design: developing mobile application accessibility guidelines for people with visual impairment. In *Proceedings of HCI Korea*. Hanbit Media, Inc., 31–38.

[64] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. Piscataway, NJ, 133–142.

[65] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*. 91–99.

[66] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O Wobbrock. 2018. 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*. ACM, 119–130.

[67] Ch Spearman. 2010. The proof and measurement of association between two things. *International journal of epidemiology* 39, 5 (2010), 1137–1150.

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[69] Ramakrishna Vedantam, C Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4566–4575.

[70] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3156–3164.

[71] Fahui Wang. 2012. Measurement, optimization, and impact of health care accessibility: a methodological review. *Annals of the Association of American Geographers* 102, 5 (2012), 1104–1112.

[72] Xu Wang, Chunyang Chen, and Zhenchang Xing. 2019. Domain-specific machine translation with recurrent neural network for software localization. *Empirical Software Engineering* 24, 6 (2019), 3514–3545.

[73] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 226–237.

[74] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

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

[76] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. 2016. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology* 21, 1 (2016), 114–123.

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

[78] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. 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*. ACM, 609–621.

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

[80] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: Vision-Based Linting of GUI Animation Effects Against Design-Don't Guidelines. In *42nd International Conference on Software Engineering (ICSE '20)*. ACM, New York, NY, 12 pages. https://doi.org/10.1145/3377811.3380411

[81] Hui Zhao, Min Chen, Meikang Qiu, Keke Gai, and Meiqin Liu. 2016. A novel pre-cache schema for high performance Android system. *Future Generation Computer Systems* 56 (2016), 766–772.