

# Towards Complete Icon Labeling in Mobile Applications

Jieshan Chen\*  
Australian National University  
Canberra, Australia  
jieshan.chen@anu.edu.au

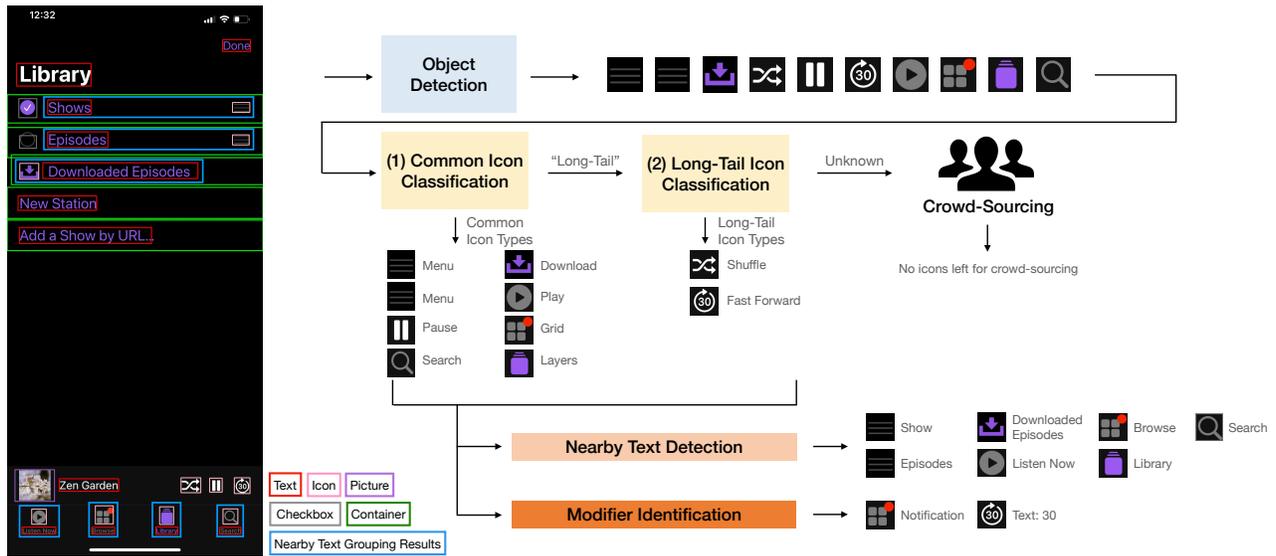
Amanda Swearngin  
Apple  
Seattle, WA, USA  
aswearngin@apple.com

Jason Wu  
Apple  
Seattle, WA, USA  
jason\_wu2@apple.com

Titus Barik  
Apple  
Seattle, WA, USA  
tbarik@apple.com

Jeffrey Nichols  
Apple  
Seattle, WA, USA  
jwnichols@apple.com

Xiaoyi Zhang  
Apple  
Seattle, WA, USA  
xiaoyiz@apple.com



**Figure 1: Flowchart of our system: It detects UI elements from an app screenshot. For each icon detection, it classifies whether the icon belongs to a common icon type. Otherwise, it uses a few-shot classification method to assign a long-tail icon type. To provide additional information, it leverages heuristics to find the icon’s nearby text, and locates any modifier symbol inside the icon. In this screen from the Apple Podcasts app, our system provides labels for all 16 icons with additional information (e.g., “Listen Now” text near the *Play* icon, a “Notification Dot” modifier inside the *Grid* icon).**

## ABSTRACT

Accurately recognizing icon types in mobile applications is integral to many tasks, including accessibility improvement, UI design search, and conversational agents. Existing research focuses on recognizing the most frequent icon types, but these technologies fail when encountering an unrecognized low-frequency icon. In this paper, we work towards complete coverage of icons in the wild. After annotating a large-scale icon dataset (327,879 icons)

from iPhone apps, we found a highly uneven distribution: 98 common icon types covered 92.8% of icons, while 7.2% of icons were covered by more than 331 long-tail icon types. In order to label icons with widely varying occurrences in apps, our system uses an image classification model to recognize common icon types with an average of 3,000 examples each (96.3% accuracy) and applies a few-shot learning model to classify long-tail icon types with an average of 67 examples each (78.6% accuracy). Our system also detects contextual information that helps characterize icon semantics, including nearby text (95.3% accuracy) and modifier symbols added to the icon (87.4% accuracy). In a validation study with workers ( $n = 23$ ), we verified the usefulness of our generated icon labels. The icon types supported by our work cover 99.5% of collected icons, improving on the previously highest 78% coverage in icon classification work.

\*Work done at Apple.



This work is licensed under a Creative Commons Attribution International 4.0 License.

CHI '22, April 29-May 5, 2022, New Orleans, LA, USA  
© 2022 Copyright held by the owner/author(s).  
ACM ISBN 978-1-4503-9157-3/22/04.  
<https://doi.org/10.1145/3491102.3502073>

**ACM Reference Format:**

Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. 2022. Towards Complete Icon Labeling in Mobile Applications. In *CHI Conference on Human Factors in Computing Systems (CHI '22)*, April 29–May 5, 2022, New Orleans, LA, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3491102.3502073>

**1 INTRODUCTION**

Icons are an essential part of mobile user interfaces (UIs), and have been found to be the second most frequent UI element type after text in mobile applications (apps) [35]. Unfortunately, unlike text elements, icons are not accessible by their nature and typically require a separate label to be specified by the developer in order to become explainable to users of accessibility technologies. Ross et al. [28] conducted the first large-scale analysis of the accessibility of mobile apps and found that more than half of clickable icons are unlabeled. Another study showed that due to rapid application iteration speed and lack of awareness of accessibility issues, more than two-thirds of icons and image-based buttons are missing labels across 77% of 10,408 Android apps [8]. In some cases, the lack of an explicit label on the icon may be offset by a nearby companion element that provides a label or explanation, but our analysis in this paper shows that more than half of icons are standalone (Section 4.3).

To address this problem, systems have been built to provide icon labels when they are not available [8, 10, 25, 36]. Some ask humans to crowdsource labels, which can be error-prone and time-consuming. For example, Zhang et al. [36] proposed an interaction proxy to allow end-users to manually add labels to icons and perform runtime repair. More recent work has explored using machine learning methods [10, 25, 35] to generate icon labels based on their pixels. These systems apply image classification models to identify different icons types, and an increasing number of icon classes are supported in successive models. A weakness of this approach is that they are only able to classify icons of the supported types, and are unhelpful for understanding an icon not in the known set. Although some work leverages contextual information [8, 22, 31] to support more icon types with improved accuracy, the context—for example, the view hierarchy or source code—is sometimes incomplete or not accessible by icon recognition services [18, 19, 33]. The APIs to access view hierarchy may also change or become unavailable [30]. Furthermore, Zhang et al. [35] found that 59% of screens contain some UI elements that are not in the accessibility hierarchy, and 94% of apps in the dataset have at least one such screen.

Instead of using the unreliable view hierarchy, our approach leverages pixel-based context on app screen, including nearby text of icons and modifiers (secondary symbols) inside icons. As observed in Section 3.3.3, more than half of icons are accompanied by meaningful nearby text. Previous work [27] indicated that the nearby text can be the most relevant one among all contextual information. Another important context, modifiers, may change the meaning of icons. For example, in Figure 10(b), adding the *Disabled* modifier to the *Camera* icon completely reverts its meaning.

In order to characterize the icon recognition problem more comprehensively, we start our work by examining a large dataset<sup>1</sup> of

327,879 icons from iPhone apps extracted from screenshots in the AMP dataset [35]. We used crowdsourcing to annotate the icons with an initial list of 90 pre-defined icon types, and collected open-coding labels from annotators for icons identified outside of those types. We used k-means clustering to group the icons with open-coded labels and identified 339 more icon types. Some of these were more common than any in our pre-defined set, so we designated them as common types—yielding a total of **98 common icon types** and **331 long-tail icon types**. We found that 92.8% of the icons could be covered by the 98 common icon types. In the remaining 7.2% of icons, 0.5% icons were so uncommon that we could not classify them into a type (e.g., they only appear in one or two apps). In examining these long-tail icons, we found that while they account for less than 10% of all icons, they often expose important app functionality and should be supported by icon recognition systems. For example, while the *Truck* (0.04%) and *Shuffle* icons (0.06%) (Figure 4) occur less frequently than other common icon types—such as menu (2.8%) and search (4.4%) (Figure 1)—these two icon types are often used in delivery and music apps and provide access to core functionality in these apps.

To generate labels for both common and long-tail icons in a broad range of scenarios—for example, when lacking access to accessibility metadata and the view hierarchy—we designed an end-to-end system that takes only screenshot pixels as input. After detecting UI elements in a screenshot [35] and extracting each icon, we ran an image classification model to check if an icon belongs to a common icon type. Otherwise, we used a few-shot classification method to assign a long-tail icon type, which utilizes the prior knowledge learned from the common icon types and some long-tail icon examples. To provide additional information, we found the icon’s nearby text by heuristics and examine the modifier symbols in an icon. We identified seven common modifiers (Figure 10) and synthesized a modifier dataset to assist recognition. We applied OCR to recognize the text modifier and trained an object detection model to recognize the remaining six modifiers. If these steps fail to generate meaningful labels, crowdsourcing can be introduced to create icon labels.

We evaluated the proposed system by first examining each module individually, then running modules end-to-end, and finally conducting a validation study with 23 workers to examine the overall quality of our annotation and predicted labels. Our common icon classification model has a performance of 96.3% accuracy, and our long-tail icon classification model achieves 78.6% accuracy. Within 500 randomly sampled UIs, our nearby text detection module achieves 95.3% in accuracy in identifying relevant nearby text. Our modifier identification model also reaches 87.4% accuracy across 462 icons. The usefulness of our annotation and the proposed system is further confirmed by a validation study on 2,064 icons, with 96.9% annotations and 80.3% predictions considered as useful labels by at least one worker.

In this paper, we make the following contributions:

- An analysis of a large dataset of 327,879 icons we extracted and annotated from iPhone apps, identifying a highly uneven distribution that 98 common icon types contain 92.8% of icons, while 6.7% of icons belong to 331 long-tail icon types; 0.5% are too niche to be classified.

<sup>1</sup>Icons and screenshots in all figures either originate from Apple apps or are mock-ups representative of apps in our dataset constructed using public domain icons.

- A pixel-only method that generates icon labels by classifying common icon types, identifying long-tail icon types that have few examples, leveraging nearby text, and recognizing modifier symbols inside icons. The icon types supported by our method cover 99.5% of collected icons.

## 2 RELATED WORK

Our work builds upon mobile app UI datasets and uses this data as an important input to our icon recognition methods. Both mobile app UI datasets and existing icon recognition methods provide important context for our work.

### 2.1 Analyses of Mobile App Icon Datasets

Researchers have collected datasets to improve the understanding of UIs and their semantics. Deka et al. [9] collected a large-scale UI design dataset, called Rico, that contains over 72k screenshots with view hierarchies from 9,772 Android apps. Following this work, Liu et al. [25] extracted icons from the Rico dataset. They defined several heuristics to obtain the bounds of icons from view hierarchies, in order to crop icons from screenshots. They identified 135 common icon types through an iterative open coding, and annotated 73,449 icons extracted from Rico. The limitations of this dataset include: 1) 18% of icons are too niche to belong to one of 135 common icon types and thus are not labeled; 2) view hierarchies may not match their screenshots in more than half of screens [21] and therefore cannot reliably locate icons. As a result, this dataset covers only a portion of existing icons on the Android platform. Through manual inspection, recent research has also highlighted noise and other quality issues within the Rico dataset [18, 19].

To increase coverage of icons, Chen et al. [8] leveraged developer-provided content descriptions as the icon label. From 7,594 apps, they collected labels of 19,233 image-based buttons, which include both common and long-tail icons types. However, due to mismatched view hierarchies, icons may be associated with the wrong labels. In addition, content descriptions may be uninformative or low quality [28]. To solve poorly matching view hierarchies, Zang et al. [33] re-annotated the Rico dataset with a crowdsourcing approach. From app screenshots, the crowd workers drew bounding boxes and assigned one of 29 types to each icon. Without relying on view hierarchies, they annotated 137,282 icons, which are 40% more icons than in previous work [25].

In addition to extracting icons from mobile app datasets, Feng et al. [10] collected a large-scale dataset of 41,000 icons from an existing sharing platform for icon design. As designers use different ways to express the same icon concept, the researchers utilized an association rule mining method [1] to find frequent co-occurring labels, and then manually identified 100 icon categories.

Our dataset is similar in form to datasets considered above, although it is derived from iOS rather than Android. For icons extracted from the AMP dataset [35], we used crowdsourcing and automatic clustering methods to annotate labels for the vast majority of icons in our resulting dataset. The total number of icon classes that we consider is larger than any work above, and includes 429 classes spanning both 98 common and 331 long-tail icon types.

### 2.2 Icon Recognition Methods

Recognizing icons can benefit many tasks, including accessibility [8, 35], UI design search and generation [5, 7, 37], app security [32], and conversational agents [20].

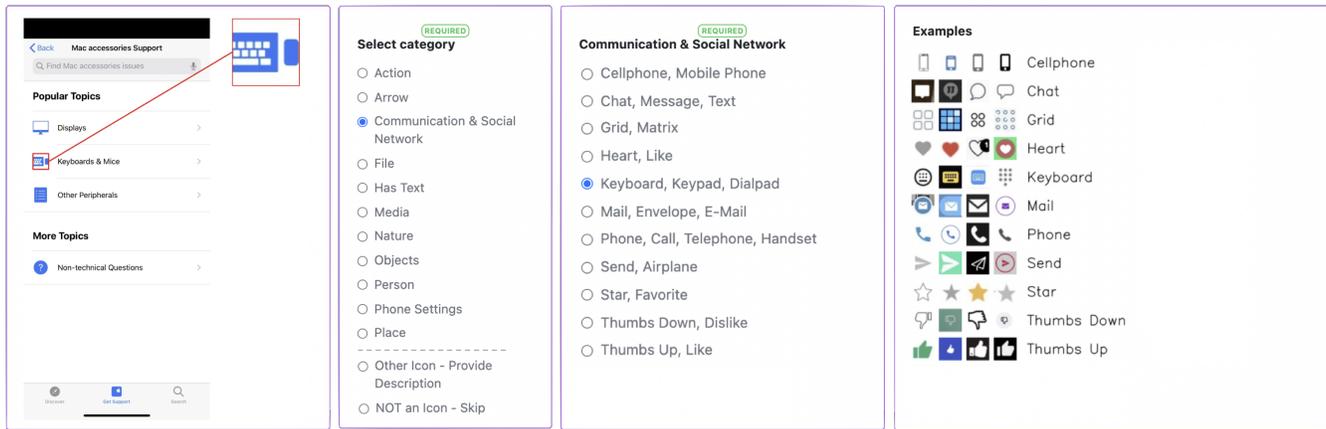
To identify icon types from icon pixels, Liu et al. [25] adapted a convolutional neural network (CNN) architecture to train a deep learning model that classifies 99 common icon classes in Android apps. Xiao et al. [32] extracted features from icon pixels with a variant of the SIFT algorithm and then found the closest icon type by a k-nearest-neighbor-like method. To facilitate web UI development, Feng et al. [10] created a pipeline for font conversion, icon label prediction, and color detection from cropped icon pixels. Our methods also leverage icon pixels to classify icon type, but we applied image classification methods and few-shot classification methods—allowing us to support both common and long-tail icon types.

Contextual information may further support icon recognition, and previous work has accessed the view hierarchy or source code for more context around icons. Xi et al. [31] found that similar icons may reflect different intentions in different UI contexts, and that nearby text may help in distinguishing the icon context. They located contextual text by analyzing UI layout files and icon file names and fused the text with the icon pixels to classify the icon into several types. Li et al. [22] proposed widget captioning, a task to generate natural language descriptions for UI elements that are missing labels. Their multimodal inputs include the view hierarchy and screenshot pixels. LabelDroid [8] similarly framed the icon recognition problem as an image captioning task. They are able to make accurate predictions for missing accessibility labels and generate labels that have higher quality than the accessibility labels added by junior Android developers. Mehralian et al. [27] found that icon images are insufficient in representing icon labels, and proposed a context-aware label generation approach that outperforms LabelDroid [8]. They incorporated different sources of data from the view hierarchy (e.g., App Category, Activity name, Android id) to predict an icon label. Zang et al. [33] framed the problem as an object detection task and built a multi-modal pipeline that recognizes icons by leveraging the view hierarchies in addition to icon visual features. It predicts the most commonly used 29 icons in Android apps.

Our system also leverages context information (e.g., nearby text), but only uses pixel information without requiring access to app metadata (e.g., view hierarchy). Comparing with the work above, our system achieves similarly high accuracy in common icon classifications, and can also recognize long-tail icon types with few samples. In addition, our system detects nearby text and recognizes several modifiers (Figure 10) in icons to provide more semantics in labeling.

## 3 IOS APP ICON DATASET

We examined existing icon datasets [6, 10, 25, 33], and attempted to emulate their best practices while mitigating some of their limitations. In particular, we took note of the problems with determining icon bounding boxes from the view hierarchy, and chose a different method using human-defined bounding boxes. We also designed



**Figure 2:** In our icon annotation interface, we highlight an icon in the displayed screenshot. Annotators first select an icon category and then pick a pre-defined icon type. They can see four examples of each icon type on the annotation interface and find more examples in our annotation instructions. If an icon does not belong to any pre-defined icon type, annotators write a concise description. They can also skip a task if it does not contain any icon.

our annotation process so that annotators can pick a label for pre-defined icon types and write labels for other icons. This allowed us to understand the potential problems that might be associated with long-tail icon type identification and develop techniques to address them.

### 3.1 Icon Annotation

Our icons are extracted from the AMP dataset [35], a large-scale dataset that contains recently collected iOS app screens. In the AMP dataset, workers annotated a bounding box and a UI type for each UI element on every app screen. While the icons are identified in the dataset, the content of the icons are not labeled. To construct labels, we picked UI elements annotated with the “Icon” UI type and applied an additional set of annotation processes. The AMP dataset contains screens from 77,637 UIs among 4,068 top free iPhone apps in 22 app categories [35]. From this dataset, we extracted 338,343 icons from the 66,364 screens within 3,910 apps that contained “Icon” annotations.

There are 11,273 screens without any icon annotations; we manually examined a subset and found most of them to be screens with a popup dialog on blurred background, full text screens (e.g., privacy policy), or welcome / login screens that show text and a big picture to present the content. We also found that 156 apps did not contain any screen with icons, usually because the dataset only included screens for the initial welcome screens. This may have occurred because of an issue during the app crawling (e.g., could not log in without special credentials). We further explored the dataset to understand how frequently annotators may have missed annotating icons that were actually present. From screens without any icon annotations, we randomly sampled 100 screens and found that 93 screens did not contain any icons, 1 screen contained an icon in a blurred background, and 6 screens contained icons (either missed in annotation, or annotated as “Picture.” These minor flaws in the original dataset annotation [35] could be addressed in future work.

**Annotation Task:** Twenty workers annotated icon labels based on the icon image and its context on the app screen. As shown in Figure 2, for each task we showed an app screenshot and highlighted an icon inside it. Annotators either picked a pre-defined icon type, or wrote a few words as a concise icon label. They could also report if the task did not contain a proper icon—such as when the icon is occluded by another UI—or when the highlighted element was not an icon. The details of workers we recruited for data annotation and the instructions that annotators received can be found in our supplementary materials.

**Pre-defined Icon Types:** We identified pre-defined icon types from multiple sources, including previous icon-relevant work [6, 10, 25, 33], manually examining icon images in our dataset, and analyzing common text in developer provided icon labels. This led to a final list of 90 pre-defined icon types, which can be found in our supplementary materials.

**Annotation Logistics:** Each icon was annotated by two annotators; when there was a disagreement, we introduced a third annotator. Finally, we invited a QA (Quality Assurance) team to verify and correct the icon labels.

### 3.2 Annotation Results Processing

We removed annotations that were not proper icons as reported by annotators, leaving 327,879 icons. Of these remaining icons, 91.2% (298,928) belong to the 90 pre-defined icon types. We found the top three most frequent icon types to be *Back* (11%), *Right Arrow* (10%), and *Close* (7%).

We corrected missed icons belonging to the pre-defined types. In some cases, annotators forgot or neglected the pre-defined icon types and instead wrote their own labels. From the clustered long-tail icon types, we found several clusters that were the same or similar to the pre-defined types. For example, *Bag* is a pre-defined icon type, yet we found *Bag* and *Basket* clusters during our long-tail icon type processing. Icons in such clusters were reassigned to their appropriate predefined icon type category.

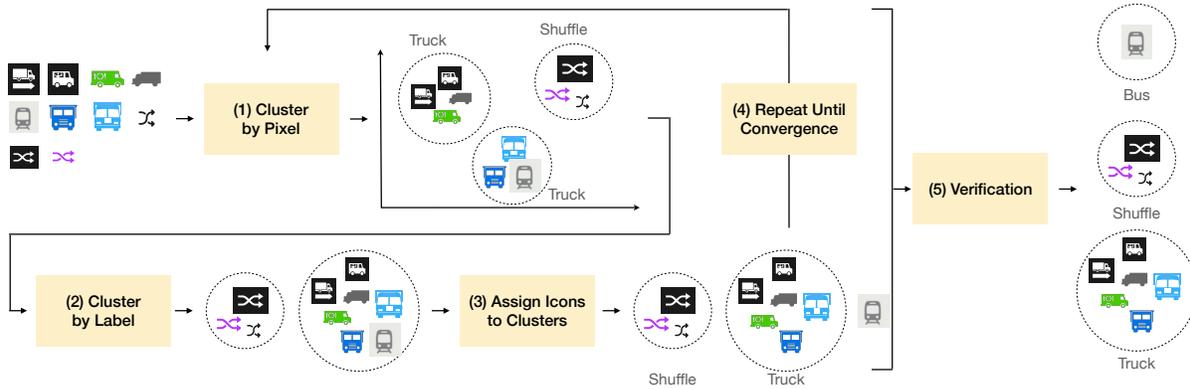


Figure 3: The clustering process consists of five steps: 1) cluster the icons by pixels; 2) compute keywords for each potential cluster and merge clusters with the same keyword; 3) assign icons to a cluster if the distance between the icon and the corresponding cluster centroid is less than a predefined threshold; 4) repeat step 1-3 until convergence; 5) let workers verify the clustering results.

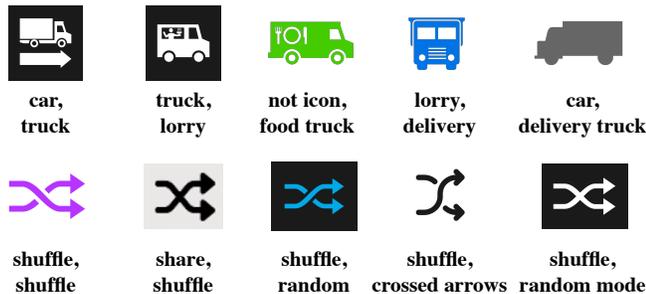


Figure 4: Examples of two long-tail icon types, each with labels provided by two annotators (delimited by comma). Annotators may provide different labels to describe the same icon. Therefore, we tokenized the result in order to cluster long-tail icon types.

We removed company logo icons. Some of annotated labels contained the keyword “logo.” Most are company logos which appear with low frequency, except for some log-in providers like Apple, Google, Facebook, and Twitter which were already in the pre-defined icon types.

For the icons outside the pre-defined types, we applied clustering to find long-tail icon types. An overview of the clustering process is shown in Figure 3, in which we leveraged both the icon pixels and the labels provided by annotators. The steps in the cluster process are:

- (1) **Cluster by Pixel:** Icon pixels were used to perform initial clustering: we extracted icon features from an image classification model trained with pre-defined icon annotations and then performed k-means clustering to group icons not assigned to the pre-defined types. To find the optimal number of clusters, we utilized the elbow method [11]. More specifically, we calculated the mean value of the distances between each point and the corresponding cluster centroid to measure the cohesion of the clustering results—a lower

score indicates a better clustering. In our case, to ensure each cluster only contains icons of the same type, we used a slightly larger number of clusters. Based on the computed scores from different numbers of clusters, we selected the value  $k = 4000$  after the elbow point as the optimal number of clusters.

- (2) **Cluster by Label:** Labels written by annotators were then used to further merge the highly-related clusters. As most labels are short—comprised of less than five words—we directly use a simple yet efficient method to further merge clusters. In each cluster, we picked the keyword with highest frequency among all annotated labels. Since our annotators might use different ways to describe the same icon (e.g., in Figure 4 top, the annotators used truck, delivery truck, and lorry), we tokenized each label, lemmatized each word to consider different inflected forms as a single item, and calculated the frequency of each word. After defining a keyword for each cluster, we merged clusters that share the same keywords, re-calculated the keywords in the new clusters, and repeated the merging process until there were no clusters with the same keyword.
- (3) **Assign Icons to Cluster:** In each new cluster, we calculated the distance between each icon and its corresponding centroid. We assigned icons to a cluster that were within a distance of  $\beta = 5.6$  from centroid. This value was chosen by observing the clustering results.
- (4) **Repeat Until Convergence:** We repeated steps 1-3 above to find more clusters until the clustering results contained mostly irrelevant icons in each cluster. We manually merged some clusters with similar keywords that clearly should have been merged but were not—for example, because lemmatization was not comprehensive enough.
- (5) **Verification:** Once we obtained our final icon types, we asked crowd workers to verify the clustering results. During verification, annotators corrected 2,265 icons. The keyword of each cluster became the long-tail icon type.

**Table 1: Statistics of our icon dataset. As the name “common” indicates, almost every screen, every app, and every app category contains common icons. Long-tail icons have lower frequency, but still appear in many screens, more than half of apps, and all app categories.**

	Common Icons	Long-tail Icons	Unclassified Icons	Total
# of Icon Types	98	331	-	429
# of Icons	304,310	22,088	1,481	327,879
# of Screens	65,906	14,376	1,234	66,138
# of Apps	3,899	2,399	739	3,904
# of App Categories	22	22	22	22

Following long-tail icon clustering, we discovered that eight long-tail icon clusters (e.g., *Library*, *Radio*) contained more icon examples than some pre-defined icons. Due to their high frequency in our dataset, we combined these eight high occurring clusters with the pre-defined icon types to create the set that we call **common icon types**.

### 3.3 Data Analysis

After processing the annotation results, we found that the 98 common icon types have 304,310 icon examples (avg = 3140, min = 409, max = 36098, std = 5404). The 331 long-tail icon types have 22,088 icon examples (avg = 67, min = 1, max = 399, std = 84). There are also 804 company logos with 3,324 examples, which we do not include in the scope of this paper as each logo is often used in only one or two apps. Next, we present some initial findings.

**3.3.1 High-Level Distribution.** From Table 1, we found:

- Every app category has apps that contain both common icons and long-tail icons.
- Almost every app (99.9%) contains some common icons, and more than half of apps contain long-tail icons.
- Almost every screen (99.6%) contains some common icons, while only 21.7% of screens contain long-tail icons.

Further inspecting Figure 5, we found a highly uneven distribution resembling a long-tail distribution. The count of all long-tail icons is similar to the count of the most frequent icon type. Therefore, the frequencies of long-tail icons are almost unnoticeable in the plot. Consequently, we used the logarithm of the distribution to better present the trend of long-tail icons, and found that the logarithm frequency of long-tail icons drops almost linearly.

**3.3.2 App Categories.** We analyzed the distribution of common and long-tail icons contained within apps aggregated across app categories. For each icon type, we counted the number of app categories in which it appears within an app. For the 98 common icon types, on average, a common icon type shows up in 21.89 app categories out of 22 possible categories. In other words, almost all common icons show up in apps in almost all app categories. In contrast, a long-tail icon type, on average, appears in apps of only 9.47 app categories. This result suggests that common icon types may support basic functionality that is needed across almost all app categories, while long-tail icon types expose specific functionality that only exists in some app categories.

As shown in Figure 6, the ratio of long-tail icons and common icons varies across different app categories (avg = 6.8%, min =

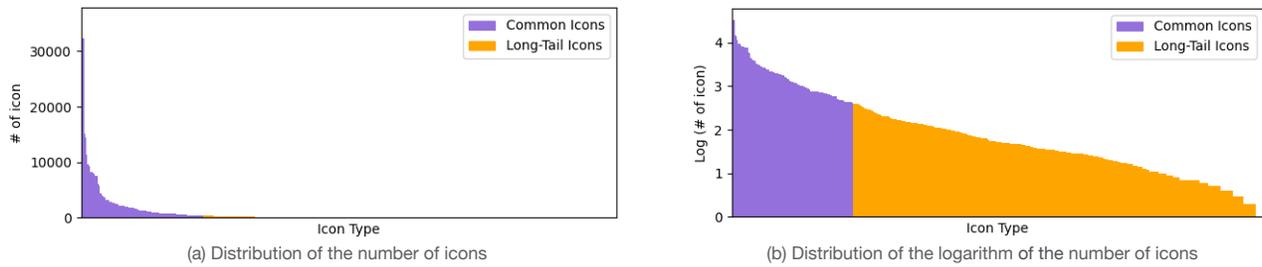
2.9%, max = 8.9%, std = 1.7%). It is worth noting that long-tail icons have a higher occurrence in Photo & Video (8.9%) and Navigation app categories (8.9%). Photo & Video apps often involve many photo-editing related icon types, such as *Crop* and *Color Filter*. Similarly, Navigation apps require many transport-specific (e.g., *Bus*) and place-related icons (e.g., *Cutlery* icons for restaurants) that belong to long-tail icon types. This finding further suggests that while common icons are prevalent across all app categories, long-tail icons are also indispensable, especially in some app categories.

**3.3.3 Properties of Icons.** Within our dataset, we observed that the semantics of icons are relevant to many factors, including their basic shapes, nearby text, modifiers (secondary symbols), and other contextual information on the app screen. In this section, we discuss the details of our observations, motivating the design of our proposed pipeline.

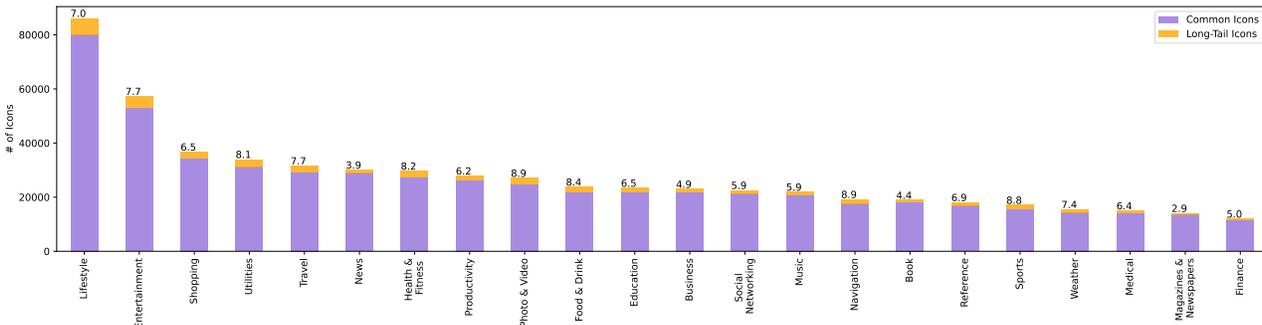
**Nearby Text:** There are three typical design patterns involving icons and text:

- (1) *Standalone* icons must indicate their functionality without requiring any nearby text. User interface guidelines [3] often recommend this design practice only for common icon types that are used across many apps. For example, in Figure 9(f), the familiar *Close* icon is used in many apps and does not need accompanying text.
- (2) *Partial* icons provide some indication of the user experience but require nearby text for the user to fully understand its meaning. For example, Figure 9(g) shows a *Comment* icon with a number nearby (the count of comments), and Figure 9(j) shows a *Play* icon next to the text “Slideshow” that completes the explanation of what tapping in that area will do.
- (3) *Duplicate* icons have the same or similar meaning as their nearby text. Although these icons seem redundant, design guidelines recommend this practice, as it can still reduce users’ cognitive load in recognizing icons. When an icon has multiple meanings in different scenarios, the nearby text determines the most suitable one. For example, in Figure 9(a), the *Error* icon is accompanied with a “Report” text, which both help users disambiguate the intent. More examples can be seen in tab bar of Figure 9(c).

We randomly sampled 500 screens (about 22 screens from each app category) and manually grouped nearby text for 2,535 icons in 500 screens. 1,183 (46.7%) icons were *standalone*, 995 (39.3%)



**Figure 5: Highly uneven distribution of icons across icon types, which resembles a long-tail distribution. The count of all long-tail icons is similar to the count of the most frequent icon type.**



**Figure 6: Bar chart of icon frequency and the percentage of long-tail icons (the number above each bar) in each app category. Each app category has similar number of apps and screens collected. The Lifestyle app category in particular contains more icons than other categories. Some app categories (e.g., Photo & Video, Navigation) have a higher percentage of long-tail icons than other categories (e.g., News, Magazines).**

groupings used the *partial* design pattern, and 357 (14%) groupings used the *duplicate* pattern. This result confirms the importance of recognizing basic shape and contextual information.

**Modifiers (Secondary Symbols):** In examining our icon dataset, we noticed that some icons are comprised of two symbols: one main symbol showing the primary concept of the icon, and another smaller symbol providing additional information. We call these second smaller symbols *modifiers*. For example, the top-left icon in Figure 10 (c) shows the folder symbol as the primary concept with a star modifier indicating that the icon might be a *Favorite Folder*. In addition, modifiers may change the meaning of icons. For example, in Figure 10(b), adding the *Disabled* modifier to the *Camera* icon completely reverts its meaning. Efficiently recognizing modifiers is also crucial for understanding icons.

From 500 sampled screens (Section 4.3), we manually identified modifiers inside icons. Among 2,535 icons, 67 (2.6%) contained a modifier symbol: short text (19), *Add* (15), *Disabled* (7), *Star* (5), *Notification Dot* (5), *Checkmark* (4), and *Clock* (2). The remaining modifiers only had one example, including *Location*, *Plot*, *Currency*, *Music*, *Lighting*, *Recycle*, *Snowflake*, *Down Arrow*, *Search*, and *Play*.

**Summary:** Informed by these findings, we designed an end-to-end system—towards complete icon labeling—by leveraging deep-learning techniques and crowdsourcing methods. We designed two classification models to recognize the basic shape of icons, a heuristics-based method to find contextual information (i.e. nearby

texts), and an object detection model to identify modifiers. Crowdsourcing will be introduced if all of the other methods fail.

## 4 SYSTEM

Figure 1 shows the flowchart of our system, which takes a screenshot as input and returns a label for each icon in the screen. First, we run an object detection model based on Zhang et al. [35] to recognize all UI elements. Then, we extract the pixels for each icon detection. Although the bounding box of the icon detection may not be exactly square (e.g., *More* icon, *Shuffle* icon in Figure 4)—as icons are nevertheless mostly square in UI design—we extend the bounding box to be a square using the larger-side length, maintaining the original center. Next, we crop the icon from screenshot using this expanded square bounding box. Maintaining a constant aspect ratio will help when applying ML methods.

For each detected and extracted icon, we run an image classification model to determine if the icon has a common icon type. Otherwise, we use a few-shot classification method to assign a long-tail icon type. We also leverage heuristics to find the icon’s nearby text, and employ an object detection model to locate any modifiers within the icon. If the icon label is still unknown after these steps, we introduce crowdsourcing method to provide a label. However, this is seldom necessary as our system achieves almost complete coverage of icons. The details of each step are described in the following subsections.

## 4.1 Common Icon Classification

An image classification model is trained to classify common icon types. We leveraged ResNet-50 [12]—which is pre-trained on the ImageNet dataset—and fine-tuned the model for icon features as described in previous work [8, 10, 27].

**Data:** In addition to the 98 classes of common icon types, the “long-tail” class is assigned to all icons not in the common icon types. When the model classifies an icon as a “long-tail” type, we apply the few-shot classification method described in the next section. For each icon type, we pick approximately 80% of examples as training data, 10% of examples as validation data, and 10% of examples as testing data. To avoid the data leakage problem [16], we split the dataset in a way that, for a given icon type, icon examples from one app will be in only one of the splits.

**Handle Class Imbalance:** As shown in Figure 5, our dataset is highly imbalanced—the most frequent icon type has 36,098 examples while the least frequent one has only 439 examples. To handle such an imbalanced dataset, we use focal loss [24] so that the weight for the “easy examples” is reduced to let the network focus on training the “hard examples.” That is, instead of giving equal weighting to all training examples, focal loss down-weights the well-classified examples.

**Model Details:** We crop each icon into a square as described in Section 4, and scale it to the input size (256x256 pixels). Each pixel keeps its RGB channels (normalized from [0,255] to [0,1]). We train our model on 4 Tesla V100 GPU for 50 epochs with a batch size of 128 and an initial learning rate of 0.001. We iteratively update the model weights using the Adam optimizer [17].

**Evaluation:** Our model achieves 96.3% accuracy on the testing dataset, and 99.0% accuracy on the training dataset. For each common icon type and “long-tail” icon type, we compute its recall and precision. The calculated macro precision is 92.4% and macro recall is 89.5% (macro = the averages of the precision and recall of each class, as reported in Liu et al. [25]).

As observed in the icon dataset and in our model predictions, icons may belong to multiple classes. For example, the first icon in Figure 7 has *Location* (because of the Location Pin symbol) as the annotated icon type. It is predicted as *Home* (because of the House symbol inside), while *Location* has the second highest confidence in prediction. Both symbols are important to show the full semantic meaning of the icon. Among 1,511 errors in our testing results, we manually observed that 182 icons have multiple important symbols. This finding motivates us to locate additional modifying symbols within an icon, which we will discuss in Section 4.4. The confusion matrix in our supplementary materials also indicates often confused common icon types (e.g., *Location* and *Map*).



**Figure 7: Examples of icons that contain two important symbols. Both symbols are important to show the full semantics of the icon.**

## 4.2 Long-Tail Icon Few-Shot Classification

As each long-tail icon type has a relatively small number of examples, we frame long-tail icon classification as a few-shot learning task. Humans can easily recognize new objects based on few samples they have seen—the few-shot learning method is built upon this observation.

**Modeling:** We adopt the prototypical model [29] to perform an episode-based training strategy to recognize long-tail icons. For each episode, we sample a subset of  $k$  icon types from the whole set of long-tail icon types and then sample  $m$  support icons and  $n$  query icons for each icon type (also called a  $k$ -way  $m$ -shot classification problem). The support icons are used to construct a prototype of the corresponding icon type, and the query icons are icons needing to be classified. By training the model through episodes, the model can learn to quickly extract the key features for each icon type. We then extract features from every support and query icon through a backbone feature extraction model, and compute prototypes for each icon type by calculating the mean features using the support icons. For each query icon, we assign the nearest prototype’s icon type as the predicted icon type. This method may alleviate overfitting issues, which are common when training data is limited. This method can also be easily generalized to new icon types, given some support samples, as the model learns to compare the difference between each prototype and the query icons instead of merely extracting the key features for each icon type.

For the backbone model, we build upon our common icon classification model, as previous work [34] shows that lower layers can capture basic features—such as vertical lines and circles—that are shared across different icon types. We remove the final fully connected layer, add one fully connected layer, and train the last two fully connected layers to enable the model to quickly extract specific features from any icon type instead of the predefined set of icon types. We use cross-entropy loss as our loss function. When training the model, we fix  $k = 50$ ,  $m = 2$ ,  $n = 20$  for batch training to force the model to extract features from few examples. We experimented with different combinations of these values but found no obvious differences in the clustering results. For inference, we sample all icons in each icon type in the training dataset to compute prototypes and test on the testing dataset. We calculate the Euclidean distance between prototypes and the query icon to decide the nearest prototype. Other dissimilarity metrics, like Mahalanobis distance, could also be substituted here to train the model and perform inference [29]. For each icon type, icons from the same app will only exist in one split. Since 47 icon types only appear in one app, we do not train or evaluate these icon types—the support and query icons in these icon types are highly similar or exactly the same.

**Data:** To train and evaluate the long-tail icon dataset, we use the corresponding “long-tail” parts in the splits in Section 4.1. If an icon type only has examples in two apps, we will keep icons in one app in training dataset and icons in another app in testing dataset.

**Evaluation:** We considered two baselines. The first baseline directly used features extracted from our common icon classification model without fine-tuning (termed kNN). Another baseline is the Relation Network [25], which models the problem as a regression

**Table 2: Evaluation of three few-shot learning methods on long-tail icons. Prototypical outperforms two baseline methods in every metric.**

	Precision	Recall	F1	Accuracy
kNN	67.3%	78.0%	66.6%	68.5%
RelationNet	71.6%	78.3%	70.2%	74.6%
<b>Prototypical</b>	<b>75.7%</b>	<b>80.7%</b>	<b>74.5%</b>	<b>78.6%</b>

problem. It concatenates mean features of support icons and features of the query icon, and uses another convolution and fully connected layers to learn a relation score.

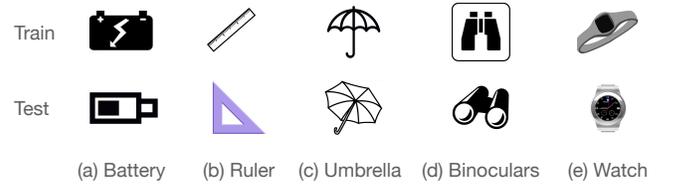
We adopted precision, recall, F1-score and accuracy as our metrics. Unlike the typical evaluation strategy that calculates average results from a number of randomly sampled episodes—including support and query icons—we treated all icons in our test dataset as the query icons and obtained support icons from the training dataset. We chose  $m = 30$  support icons when performing inference as we found this number achieved the best results on the validation dataset.

Table 2 shows the results of all models. Our model achieved 78.6% accuracy, 5.3% and 14.7% higher than the performance of the Relation Network and kNN models. All models performed well in detecting some simple icon types when the icons in the training dataset are highly similar to the icons in the testing dataset, with only minor changes. For example, the *Male Sign* icons and *Live View* icons in (Table 3-E1/E2) only have minor differences in colors and line widths compared with icons in the training dataset. While the kNN model learned to extract common patterns appearing in common icon types, it did not generalize well to unseen feature types. For example, in Table 3-E3, the *Shield* icon has two colors on two sides. Because this pattern exists in the *Contract* type and never appears in common icon types, the model was confused by the existence of this feature. In contrast, both the relation network and our prototypical network efficiently recognized the Shield icon.

We further analyzed the failure cases and observed that for some icon types, the training dataset has a different distribution from the testing dataset. The examples were either captured from different angles of the same object or had different levels of abstraction of the real object. For example, the *Battery* icon in Figure 8(a) is captured

**Table 3: Examples of icon predictions in few-shot learning methods. Baseline methods provide several wrong predictions (red).**

	E1	E2	E3	E4	E5
					
kNN	Live View	Male	Contrast	Light Bulb	Pizza
RelationNet	Live View	Male	Shield	Female	Thumbs Down
Prototypical	Live View	Male	Shield	Cup	Pin

**Figure 8: Example icons that depict the same concept from different angles or have different levels of abstraction. Long-tail icons may have such different distributions between training dataset and testing dataset.**

at a different abstraction level, and the *Ruler* icon in Figure 8(b) are actually two different kinds of rulers. In these cases, it is hard for models to correctly predict the right icon type. The details of per-class results and the confusion matrix can be found in our supplementary materials.

### 4.3 Grouping with Nearby Text

Nearby text may contain useful information for icon labeling, and thus we explored heuristics for grouping that nearby text to provide a better icon label. When our system identifies a nearby text, it appends the text to the classification label (if they are not the same) to provide more information to users.

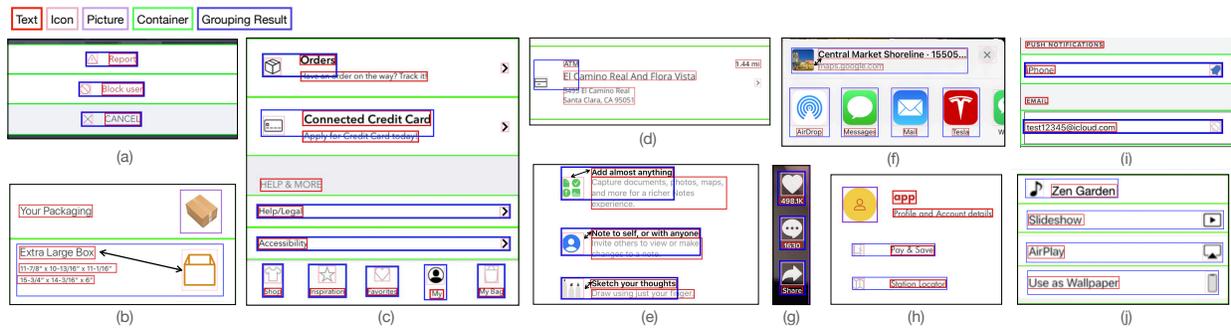
Based on observations in Section 3.3.3, from our UI element detection results, we find elements with the icon, text, and container types in our UIs and group the icons with their nearby text. Containers are a special type that often show a clear visual boundary and contain one or more UI elements.

**When an icon is within a container**, we apply a set of heuristics based on the number and types of contained icon detections and text detections:

- If the container only contains one icon and one text element, we consider that text to be the nearby text. For example, in Figure 9(a), the *Disabled* icon is grouped with “Block User” text.
- If the container contains one icon and several left-aligned text elements, we consider the first element to be the label for the icon. For example, in Figure 9(b) bottom, the *Box* icon is grouped with “Extra Large Box” text.
- If the container has several icons and several text elements, we first calculate the distance between each icon and each text element and take the pair which has the shortest distance. For example, in Figure 9(d), the *Wallet* icon would be grouped with the “ATM” text and the *Right Arrow* icon does not get any nearby text.

**When an icon is not within any container**, we leverage the spatial relationship between the icon and text detections (e.g., alignment, distance) to infer grouping:

- We first find all nearby text candidates within a distance of  $\alpha$  pixels from the icon. We obtain empirical threshold  $\alpha = 1.25 * \max(\text{icon\_width}, \text{icon\_height})$  after observing 100 screens.
- If only one text candidate has X or Y overlap with the icon, we pick that as the label. For example, in Figure 9(g), the *Chat*



**Figure 9: Examples of nearby text grouping.** Based on container detections (green) and UI spatial relationship, our heuristics groups each icon (pink) with a nearby text (red), if any. The groupings are visualized in blue.

icon is grouped with the “1630” text in the same column; in Figure 9(h), the *Map* icon is grouped with the “Station Locator” text in the same row.

- If there are multiple text candidates that have X or Y overlap with the icon, we pick the text element that is first in the reading order. For US English systems, we pick the top-most and left-most element. This approach was chosen after examining 100 screens. For example, in Figure 9(e), the green icon has two text detections in the same row, and should be grouped with “Add almost anything” text.

**Evaluation:** To evaluate the performance of the nearby text grouping, we used the annotated data from Section 3.3.3. Our heuristics achieved 92.8% precision, 98.7% recall, 95.6% F1 score, and 95.3% accuracy.

#### 4.4 Identifying Modifier Symbols Within an Icon

To further improve our icon labeling, we explored detecting the secondary modifier symbol. When our model detects a modifier inside the icon, we append the modifier label to the classification label (e.g., *Camera, Disabled*) to provide more information to users.

**Type of Modifiers:** As observed in Section 3.3.3, there are many modifiers we detected in the sampled icons. In the scope of this paper, we pick the top 7 modifiers (shown in Figure 10) to demonstrate our system. It is possible to support more modifier types with additional data.

**Synthetic Dataset:** Very few icons include modifier symbols (only 2.6%), and each symbol has very low frequency. Therefore, we created a synthetic dataset that contains enough data to train a model (shown in Figure 11). Adding modifier examples on existing icons seems a straightforward idea, but we need to take into account the following considerations:

- **Color:** We removed the synthetic examples that have a modifier in a similar color as its surrounding icon pixels, as the modifier would be invisible.
- **Position:** We observed the pattern of a modifier’s position in icon examples—the *Disabled* modifier often covers the whole icon, and the *Notification Dot* modifier often appears on the top-right.

- **Relative Size:** *Disabled* modifiers often have a width between 80% to 100% of the icon width, and *Notification Dot* modifiers have a width between 7% to 20% of the icon width. The remaining modifiers have a width between 25% to 60% of the icon width.

- **Examples:** To create a short text example, we randomly picked one word from the text detections in the dataset [35]. To create a *Notification Dot*, we created a solid circle with random color. To create a *Disabled* symbol, we drew a diagonal line, and sometimes also added a circle. Other modifier types are all in our common icon types; therefore we picked 500 examples of each modifier from common icon dataset, and applied the flood fill algorithm [13] to remove the background color of these examples.

In total, we synthesized 35,389 examples of each modifier. Figure 11 shows some examples of the synthesized icons.

**Modifier Identification Model:** Text modifiers can be recognized by OCR [4]. For the remaining 6 modifiers, we experimented with both image classification and object detection models to detect modifiers inside icons. Our image classification (**IC**) model uses MobileNetV1 [14], and our object detection (**OD**) model applies the SSD (Single Shot MultiBox Detector) [26] model with MobileNetV1 [15] as the backbone. Since the size of modifiers are relatively small (compared with icons), we also include a feature pyramid network (FPN) [23] that uses a pyramidal hierarchy of deep convolutional networks to extract the image features. Each model is trained on 4 Tesla V100 GPU for 100 epochs, with an initial learning rate of 0.001.

**Evaluation:** Both models achieved high accuracy on our synthetic testing dataset: the IC model achieved 96.7% accuracy, and the OD model achieved 95.2% accuracy. To evaluate the actual performance on real icons, we manually picked 463 icons from our icon dataset (half icons contain modifiers). The OD model achieved 87.4% accuracy (87% precision, 82% recall, and 84% F1 score), which outperformed the IC model (84.4% accuracy, 90% precision, 67% recall, and 74% F1 score). We also found that the IC model performed worst on *Notification Dot* (small circles are common inside icon, and may not provide a strong signal in classification), while the OD model performed worst on *Disabled* modifiers, most likely because it is challenging for OD models to handle objects that occupy the full image.

# Modifiers - noun project

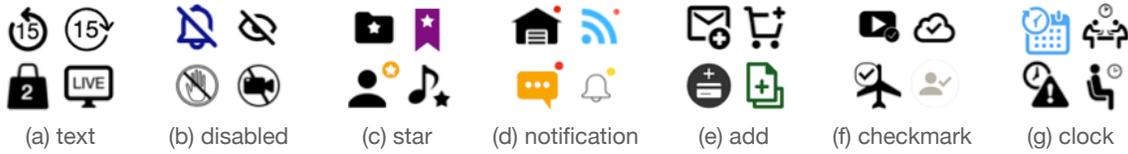


Figure 10: Example icons that contain top seven modifier symbols.

## Modifiers

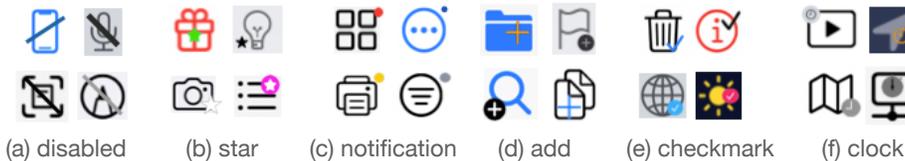


Figure 11: Examples of the synthesized icons. We augmented modifier symbols on existing icons. Note that we did not have to augment text modifiers, as we used OCR [4] to recognize text in icon.

## 5 EVALUATION

In the previous section, we evaluated each step of our system individually. In this section, we evaluated our system as a whole. We performed an end-to-end icon recognition from screenshot pixels, and conducted a validation study with workers to rate the usefulness of labels.

### 5.1 Evaluating End-To-End Icon Recognition

We evaluated the overall performance of our classification model on “imperfect” results from object detection models and also documented the errors that propagated from each step.

**Dataset:** We used the same test set from Section 4.1 and Section 4.2 to avoid the data leakage problem. In total, we obtained 32,989 test samples (30,352 common icons and 2,637 long-tail icons).

**Procedure:** From UI detection results, we cropped all icon detections, and attempted to match with overlapping icon annotations in our testing dataset. When an icon detection and an icon annotation had any overlap, we used the annotation label as the ground truth label for that icon detection. We cropped all matched icon detections into a square, as described in Section 4. We ran each icon through our common icon classification model; if the prediction result was “long-tail” or the confidence was lower than predefined thresholds (empirically defined using the validation dataset), we ran the few-shot long-tail classification method to find a label.

**Object Detection.** 93.8% of testing icons were successfully recognized by the UI detection model (84% precision, 95% recall and 89.2% F1.). We found that larger and more colorful icons were more likely to be predicted as “Picture,” which are 1.9% of the testing icons. We hypothesize that performance may be further improved by using heuristic-based post-processing of Picture detections.

For the remaining 4.3% missing icons, we noticed two situations apart from model errors. First, some icons appear in the background of UI and should ideally not be detected. However, our icon dataset still included them. Second, a portion of these icons are the top-left *Return to previous app* icon supported by iOS, which enables the user to return to the previous application: this is not an icon from the app itself.

**Icon Classification.** We cropped all matched icon detections into a square, as described in Section 4. We ran each icon through our common icon classification model. If the prediction result was “long-tail,” we ran our few-shot long-tail classification method to find a label. Among all matched detections, our system achieved 90.7% accuracy.

For icons in common icon types, our system correctly classified 91.1% of them; for icons in long-tail icon types, our system correctly classified 79.5% of them. Both are slightly lower than the results in Section 4.1. This indicates that our models are robust to bounding boxes from the object detection results, which may be less accurate than the bounding boxes from annotations. We also noticed that the precision of long-tail icons was lower (60%), which is due to our higher confidence thresholds for common icon types—causing the common icon classification model to send some lower-confidence common icons into long-tail icon classification. Detailed results can be found in our supplementary materials.

### 5.2 Evaluating the Usefulness of Icon Labels

We further confirmed the usefulness of the labels generated by our system with a validation study. To reduce bias, we recruited 23 workers who were not involved in the previous icon label annotation task. Only people without disabilities participated in the study, even though the primary motivation of our work is accessibility. In this case, we wanted to verify the accuracy of our system with people who were able to perceive the icons directly. Future work may involve additional studies with blind or low-vision screen reader users in the context of actual UI and accessibility experiences.

**Icon Dataset:** For each icon type, we randomly picked five icons from our testing dataset. For a given icon type, we picked icons from the different apps, so that the icons have dissimilar designs. In total, we obtained 2,064 icon examples (490 common icons and 1,574 long-tail icons<sup>2</sup>).

**Icon Label Generation:** We prepared the following four labels for each icon using its screenshot pixels:

<sup>2</sup>33 long-tail icon types have less than five examples in the testing dataset, and therefore we take all of their available examples.

Only the star modifier

- (1) **The annotated icon type** was extracted from its previous annotation. This helps us understand the quality of our clustered icon types, and the label usefulness when our model achieves perfect accuracy.
- (2) **The predicted icon type** was recognized by the end-to-end evaluation method in Section 5.1, without using existing annotation bounding boxes.
- (3) **The nearby text** was obtained by running nearby text grouping heuristics on the screenshot UI detection results. The text content was then extracted by OCR API [4]. We found nearby text in 1,264 (61.2%) icons.
- (4) **The modifier symbol** was predicted by our object detection model. Only 50 (2.4%) icons had modifiers detected inside, which is similar to the low frequency noted in Section 4.4.

**Procedure:** For each icon, a worker rated the usefulness for each of the four labels mentioned above. Each icon was rated by two workers. For the annotation labels and prediction labels, we adopted a three-point Likert-type scale for usefulness, with 1—not useful, 2—somewhat useful, and 3—very useful. For nearby text and modifiers, we asked whether the additional information is relevant to the icon. We showed the corresponding UI screenshot for each icon to help workers better understand the icon. The rating interface can be found in supplemental materials.

**Result:** For these four labels, we first calculate Cohen’s  $\kappa$  to measure the agreement between two ratings, and then elaborate the detailed scores for each label. We considered two strategies: a relaxed strategy and a strict strategy. For the relaxed strategy, we took the best rating for each icon. For the strict strategy, we took the worst rating.

- **Annotated Icon Type:** Cohen’s  $\kappa$  for annotation labels was 0.28, which indicates a fair agreement between two ratings. Under the relaxed strategy, 97.19% labels were considered as Very Useful, and only 0.73% as Not Useful. While different people may have different points of view on each label, this shows the upper-bound of the usefulness of our icon dataset. Under the strict strategy, we still had 88.23% labels considered as Very Useful, 7.61% as Somewhat Useful and only 3.39% (70/2064) as Not Useful. Among these "Not Useful" icons, we found that 52.8% (37/70) icons had two extreme labels: as one rater thought it was Very Useful, while another thought it was Not Useful. These results validate that our annotation label can effectively help end-users understand the meaning of icons.
- **Predicted Icon Type:** For the predicted label, we found Cohen’s  $\kappa$  to be 0.66, which indicates a substantial agreement between the two ratings. We found that 82.75% labels were considered Very Useful, and 13.81% (285/2064) as Not Useful under the relaxed strategy. Among those 13.81% Not Useful labels, 89.9% (256/285) icons were predicted inaccurately, which indicates the improvement room of our classification models. For the strict strategy, only 5.86% more labels were considered as Not Useful, with the rest 80.33% icons (72.22% Very Useful, 7.61% Useful) provided meaningful labels to end-users. This result confirms the usefulness of our classification models.

- **Nearby Text:** As some icons do not have nearby texts, we removed the ratings for these icons. Cohen’s  $\kappa$  for nearby text was 0.67, which indicates a substantial agreement between two ratings. Among these results, 91.85% were considered as Relevant to the icon under the relaxed strategy, which is consistent to our evaluation in Section 4.3. The portion of the Relevant rating was lower under the strict strategy, with a percentage of 85.05%. This result sheds light on leveraging the nearby text-to-assist icon recognition models, which we consider as future work (Section 6).
- **Modifiers:** The inter-rater agreement was also substantial for the modifier symbols, with a Cohen’s  $\kappa$  of 0.73. Only 38% symbols were considered as relevant in the relaxed strategy. We noticed that among these detections, 15/50 are *Disabled* symbols, and 14 of them are wrong predictions due to some icons have content similar to some common modifiers. As discussed in Section 4.4, the object detection model performs worst in detecting this *Disabled* symbol—a better way to synthesize this symbol may be needed.

In summary, 96.61% annotation label, 80.33% prediction label, 91.85% nearby text and 38% modifier symbols were considered as useful or relevant for at least one rater.

## 6 DISCUSSION AND FUTURE WORK

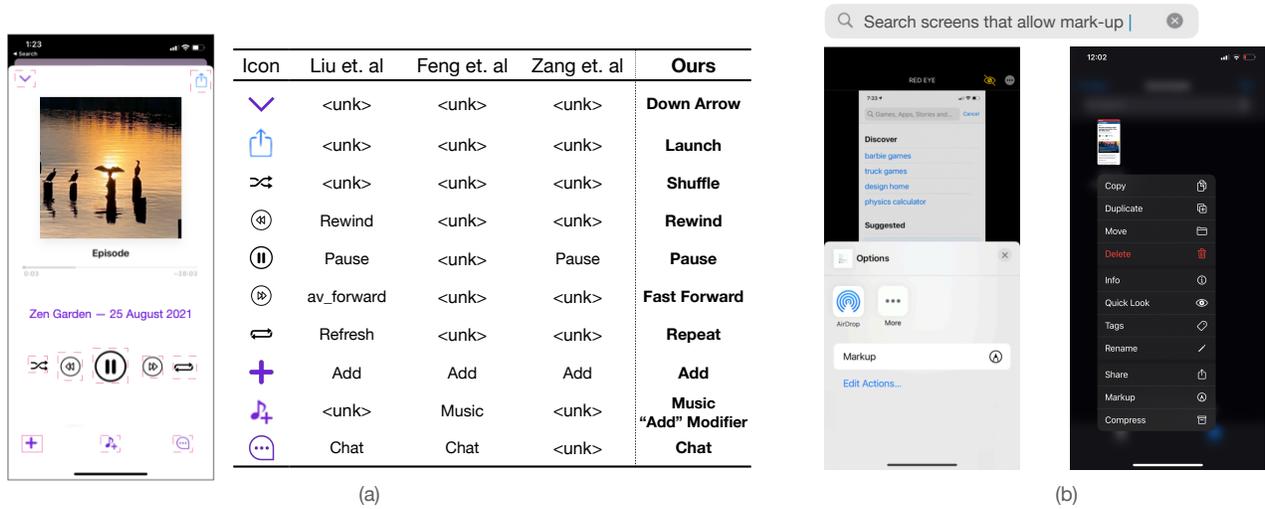
Several applications may benefit from a higher coverage of icon labeling.

**Accessibility Support for Screen Readers:** When an icon is not labeled in an inaccessible app, our system may add an accessibility label. While users without disabilities enjoy the visual appearance of icons, users with visual impairments may be impacted by missing alt-text or content descriptions for the icons if they use screen readers in inaccessible apps [2, 28]. Figure 12(a) shows a media player screen that has icons without text labels for minimalist design: this is inaccessible when developers forget to add alternative text. Compared to other icon recognition work,<sup>3</sup> our system supports common icons, long-tail icons, and even an *Add* modifier inside the *Music* icon.

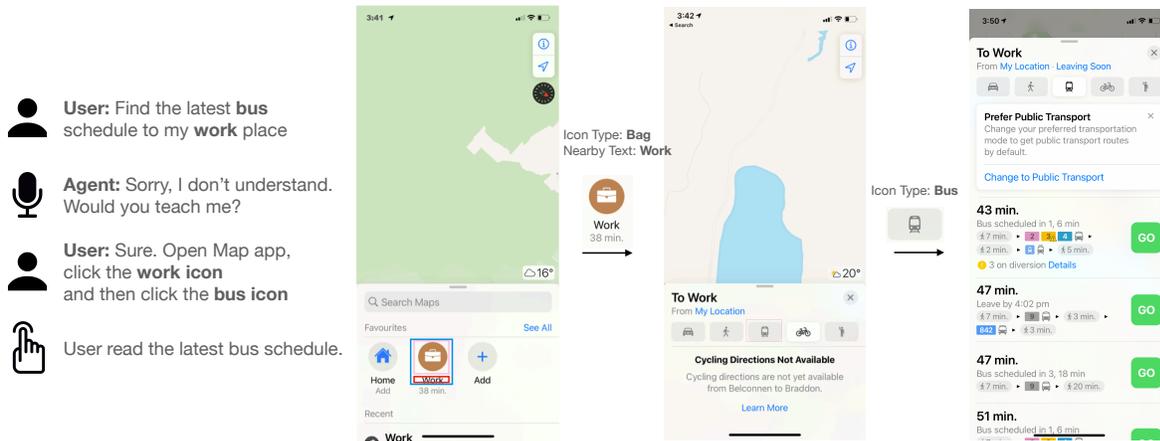
**Natural Language Based UI Search:** When designers share UI designs as images within online platforms, our pixel-based system allows search in those UI screenshots without view hierarchy information. For a certain icon type, we can provide a set of example icons, and a gallery of UI designs that contain this icon type. As seen in Figure 12(b), designers may use natural language to search relevant UIs to find some inspiration. With better coverage of icon labeling in a large-scale UI dataset, we can support more icon types in designer’s query.

**Assisting Conversational Agent:** When users interact with conversational agents, they sometimes need to refer to an icon on the screen. When some icons are unlabeled, users have to use other references (e.g., relative location) to share their intent with the agent. Our more complete icon labeling can bring a smoother experience in conversational agents. For the task of giving natural language instructions to UI actions [21], icon labeling is also crucial as it serves as an important property, “name,” of target UI. Figure 13

<sup>3</sup>Feng et al. [10] only showed 40 icon types in their paper while their method considered 100 icon types.



**Figure 12: (a) The icon labeling results on an example media player screen show the icon coverage of existing works [10, 25, 33] and our work for supporting icon annotation for screen readers. (b) Our more complete icon annotation system can support finer-grained UI design search.**



**Figure 13: Our more complete icon labeling system may help conversational agent better understand user's instructions.**

shows an example where more complete icon labeling helps the conversational agent better understand user's query.

Next, we share some limitations of our current work, and briefly discuss how to improve them in the future. One limitation is that we labeled icons only for the iOS platform. Nevertheless, these icons are commonly used in Android platform and websites.

There is also room for performance improvements. For common icon classification, we may be able to further improve its accuracy after collecting more data with further data cleaning. For long-tail icon classification, we can consider using additional contextual information from the screen to support the classification. For nearby text detection, a larger-scale annotation specific to icon-text grouping would allow us to train a grouping model. For modifier detection, we could leverage annotation with real icon data—this would

be expected to outperform the synthetic dataset. Currently we directly concatenate nearby text and modifier labels; future research may find better ways to integrate them to create more concise and accurate labels.

In addition to accuracy improvements, we would like to improve the quality of our generated icon labels. First, the same icon may have different meanings under different contexts. For example, a *Video Recorder* icon may indicate “upload video” in a video editing app, while it can indicate “start video conference” in a video conference app. With the app category and contextual information on the screen, it is possible to generate an icon label that best fits the context. Second, an icon may belong to multiple classes when it contains multiple important symbols, while our classification model only provides one predicted class. We may consider training

multiple classifiers to recognize each icon type. Third, we currently present additional information such as nearby text or the modifier symbol separately from the icon type prediction. It should be possible to combine these together to produce an enriched icon label. Finally, our work framed icon labeling as several classification tasks. Other approaches, such as Image Captioning, could be adopted or added to make our labels easier to understand.

## 7 CONCLUSION

From our large-scale icon annotation, we learned the highly uneven distribution of icon types, and automatically clustered long-tail icon types that have few examples. We have presented an approach that uses only pixel information to generate labels for both common and long-tail icons. Our technical evaluation and user evaluation demonstrate that this approach is promising. Our work illustrates a new approach towards complete icon labeling, and many applications stand to benefit from higher icon label coverage.

## REFERENCES

- [1] Rakesh Agrawal, Ramakrishnan Srikant, et al. 1994. Fast Algorithms for Mining Association Rules. In *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. Citeseer, 487–499.
- [2] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1323–1334.
- [3] Apple. 2021. Glyphs - Human Interface Guidelines. <https://developer.apple.com/design/human-interface-guidelines/glyphs/overview>. Accessed: 24/08/2021.
- [4] Apple. 2021. Recognizing Text in Images. [https://developer.apple.com/documentation/vision/recognizing\\_text\\_in\\_images](https://developer.apple.com/documentation/vision/recognizing_text_in_images). Accessed: 31/08/2021.
- [5] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery D.C.: Design Search and Knowledge Discovery through Auto-created GUI Component Gallery. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–22.
- [6] 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*. 665–676.
- [7] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-Based UI Design Search through Image Autoencoder. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–31.
- [8] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 322–334.
- [9] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschan, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. 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*. 845–854.
- [10] Sidong Feng, Suyu Ma, Jinzhong Yu, Chunyang Chen, TingTing Zhou, and Yankun Zhen. 2021. Auto-icon: An Automated Code Generation Tool for Icon Designs Assisting in UI Development. In *26th International Conference on Intelligent User Interfaces*. 59–69.
- [11] Cyril Goutte, Peter Toft, Egill Rostrup, Finn Å Nielsen, and Lars Kai Hansen. 1999. On Clustering fMRI Time Series. *NeuroImage* 9, 3 (1999), 298–310.
- [12] 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.
- [13] Dominik Henrich. 1994. Space-Efficient Region Filling in Raster Graphics. *The Visual Computer* 10, 4 (1994), 205–215.
- [14] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for Mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1314–1324.
- [15] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* (2017).
- [16] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in Data Mining: Formulation, Detection, and Avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6, 4 (2012), 1–21.
- [17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [18] Chunggi Lee, Sanghoon Kim, Dongyun Han, Hongjun Yang, Young-Woo Park, Bum Chul Kwon, and Sungahn Ko. 2020. GUIComp: A GUI Design Assistant with Real-Time, Multi-Faceted Feedback. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.
- [19] Luis A Leiva, Asutosh Hota, and Antti Oulasvirta. 2020. Enrico: A Dataset for Topic Modeling of Mobile UI Designs. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*. 1–4.
- [20] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–114.
- [21] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldrige. 2020. Mapping Natural Language Instructions to Mobile UI Action Sequences. *arXiv preprint arXiv:2005.03776* (2020).
- [22] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. 2020. Widget Captioning: Generating Natural Language Description for Mobile User Interface Elements. *arXiv preprint arXiv:2010.04295* (2020).
- [23] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. 2017. Feature Pyramid Networks for Object Detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2117–2125.
- [24] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal Loss for Dense Object Detection. In *Proceedings of the IEEE international conference on computer vision*. 2980–2988.
- [25] Thomas F Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 569–579.
- [26] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. SSD: Single Shot Multibox Detector. In *European conference on computer vision*. Springer, 21–37.
- [27] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. 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*. 107–118.
- [28] 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*. 119–130.
- [29] Jake Snell, Kevin Swersky, and Richard S Zemel. 2017. Prototypical Networks for Few-Shot Learning. *arXiv preprint arXiv:1703.05175* (2017).
- [30] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving Random GUI Testing with Image-Based Widget Detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 307–317.
- [31] Shenggu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, et al. 2019. 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*. 2421–2436.
- [32] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. IconIntent: Automatic Identification of Sensitive UI Widgets Based on Icon Classification for Android Apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 257–268.
- [33] Xiaoxue Zang, Ying Xu, and Jindong Chen. 2021. Multimodal Icon Annotation for Mobile Applications. *arXiv preprint arXiv:2107.04452* (2021).
- [34] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *European conference on computer vision*. Springer, 818–833.
- [35] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.
- [36] 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*. 6024–6037.
- [37] Nanxuan Zhao, Nam Wook Kim, Laura Mariah Herman, Hanspeter Pfister, Rynson WH Lau, Jose Echevarria, and Zoya Bylinskii. 2020. Iconate: Automatic Compound Icon Generation and Ideation. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.