Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels

Xiaoyi Zhang
Apple Inc.
xiaoyiz@apple.com

Lilian de Greef
Apple Inc.
ldegreef@apple.com

Amanda Swearngin
Apple Inc.
aswearngin@apple.com

Samuel White
Apple Inc.
samuel_white@apple.com

Kyle Murray
Apple Inc.
kyle_murray@apple.com

Lisa Yu
Apple Inc.
lixiu_yu@apple.com

Qi Shan
Apple Inc.
qshan@apple.com

Jeffrey Nichols
Apple Inc.
jwnichols@apple.com

Jason Wu
Apple Inc.
jason_wu2@apple.com

Chris Fleizach
Apple Inc.
cfleizach@apple.com

Aaron Everitt
Apple Inc.
aeveritt@apple.com

Jeffrey P. Bigham
Apple Inc.
jbigham@apple.com

ABSTRACT

Many accessibility features available on mobile platforms require applications (apps) to provide complete and accurate metadata describing user interface (UI) components. Unfortunately, many apps do not provide sufficient metadata for accessibility features to work as expected. In this paper, we explore inferring accessibility metadata for mobile apps from their pixels, as the visual interfaces often best reflect an app's full functionality. We trained a robust, fast, memory-efficient, on-device model to detect UI elements using a dataset of 77,637 screens (from 4,068 iPhone apps) that we collected and annotated. To further improve UI detections and add semantic information, we introduced heuristics (e.g., UI grouping and ordering) and additional models (e.g., recognize UI content, state, interactivity). We built Screen Recognition to generate accessibility metadata to augment iOS VoiceOver. In a study with 9 screen reader users, we validated that our approach improves the accessibility of existing mobile apps, enabling even previously inaccessible apps to be used.

CCS CONCEPTS

• **Human-centered computing → Accessibility technologies**.

KEYWORDS

mobile accessibility, accessibility enhancement, ui detection

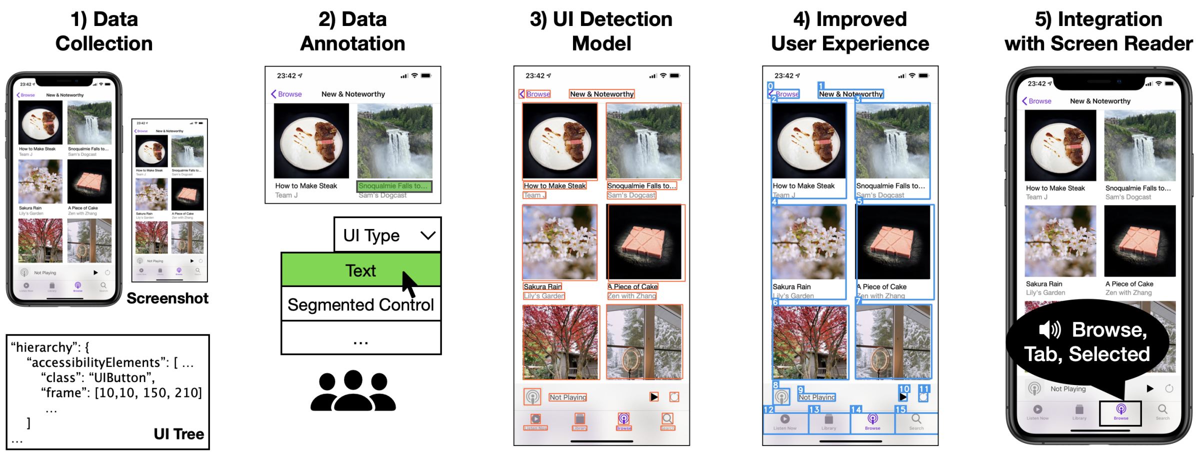
ACM Reference Format:
Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P. Bigham. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *CHI Conference on Human Factors in Computing Systems (CHI'21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3411764.3445186

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CHI '21, May 8–13, 2021, Yokohama, Japan
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8096-6/21/05.
https://doi.org/10.1145/3411764.3445186

1 INTRODUCTION

Many accessibility features (e.g., screen readers [10, 38], switch control [8, 36]) only work on apps that provide a complete and accurate description of their UI semantics (e.g., class="TabButton", state="Selected", alternative text="Profile"). Despite decades of work on developer tools and education to support accessibility across many different platforms [16, 18, 59], apps still do not universally supply accessibility metadata on any platform [42]. For example, all of the 100 most-downloaded Android apps had basic accessibility issues in a recent study [65]. The lack of appropriate metadata needed for accessibility features is a long-standing challenge in accessibility.

Apps may not be fully accessible for a variety of reasons. Developers may be unaware of accessibility, lack necessary accessibility expertise to make their apps accessible, or deprioritize accessibility. Developers often use third-party UI toolkits that can work across platforms, but have limited built-in accessibility support. For example, to make Unity apps accessible non-visually, developers either have to use a paid Unity plugin [34] that replicates a screen reader experience (catches gestures on a full-screen overlay and announces focused UI elements) or manually recreate a similar accessible experience from scratch. First party UI toolkits (e.g., UIKit for iOS) make accessibility much easier, but app developers still need to provide accessibility metadata, such as alternative text or accessibility elements for custom views [7]. Much of the focus is on creating new content and apps that are accessible, although most platforms have a substantial legacy of inaccessible content and apps that may no longer have active developer support. The myriad challenges of achieving accessibility across many different platforms and toolkits



```
"hierarchy": {
  "accessibilityElements": [ ...
    "class": "UIButton",
    "frame": [10,10, 150, 210]
    ...
  ]
...
```

Figure 1: Overview of our approach: 1) Ten workers collected screenshots and UI trees in apps. 2) Forty workers annotated the screens to provide complete metadata. 3) An on-device model (trained with annotated data) detects UI elements from pixels. 4) Heuristics (e.g., ordering and grouping) and additional models (e.g., OCR) augment UI detections to improve the user experience. 5) The generated accessibility metadata is made available to screen readers for use at runtime to improve the experienced accessibility of apps.

for so many different apps motivated us to automatically create accessibility metadata from the pixels of app user interfaces.

In this paper, we introduce a new method for providing accessibility metadata automatically from the pixels of the visual user interface. In practice, the visual interfaces for mobile apps often receive the most attention from developers and best represent an app's intended functionality. To facilitate this method, we collected, annotated, and analyzed 77,637 screens (from 4,068 iPhone apps). Using this dataset, we trained a robust, fast, memory-efficient, on-device object detection model to extract UI elements from screenshots, which achieves 71.3% Mean Average Precision. To add semantics and further improve UI detection results, we introduced heuristics to correct detections, provide navigation order, and group relevant detections; we also used additional models to recognize UI content, selection states, and interactivity. Our method creates metadata useful for a wide variety of accessibility services [8–10, 36, 38], and can be used either alone or to improve existing accessibility metadata in an app. Fig. 1 shows integration with screen readers as an example.

To demonstrate the utility of our approach, we created Screen Recognition that provides this metadata to iOS VoiceOver, using only screenshot pixels as input. In a user study with 9 screen reader users, we validated that participants could use a wide variety of existing mobile apps, including previously inaccessible ones. We also explored the limitations of our approach, and discussed future improvements as well as potential integration with developer tools. At the high level, we believe our approach illustrates a new way to start addressing the long-standing challenge of inaccessible apps on mobile and, eventually, other platforms.

Our work contributes:

- An analysis of the characteristics (e.g., UI distribution, accessibility issues) of a large dataset of 77,637 screens (from 4,068 iPhone apps) we collected and annotated.
- A robust, fast, memory-efficient, on-device object detection model to extract UI elements from raw pixels in a screenshot, which we trained and evaluated.
- Augmentation of UI detections for a better user experience. Our heuristics correct detections, provide navigation order, and group relevant detections; additional models recognize UI content, state, and interactivity.
- A user study with 9 screen reader users, who tried out apps of their choices with VoiceOver with and without our approach applied. Their feedback validates our approach can significantly improve accessibility on existing apps.

2 RELATED WORK

Our work builds from prior work on *(i)* supporting mobile accessibility, *(ii)* detecting UI elements from pixels, and *(iii)* automatic understanding of UI semantics.

2.1 Supporting Mobile Accessibility

Most mobile platforms (e.g., iOS, Android) these days contain built-in accessibility services like screen readers [10, 38]) and support for accessible input [8, 36], which allows people to use a wide variety of abilities to use them. These technologies depend heavily on the availability of accessibility metadata provided by developers to expose the underlying semantics of apps [2, 37]. When this metadata is provided, accessibility services have programmatic access to what UI elements are available, what their content contains, what state they are in, and what interactions can be performed on them. As such, this metadata is fundamental to enabling accessibility services to change the modalities by which someone can interact with UIs and support people using them in different ways. However, both previous research [65–67], and our analysis in this paper (Section 3.3), show that developers routinely fail to include this information, and many mobile apps remain inaccessible.

Prior approaches for addressing this problem include encouraging developers to fix accessibility problems through education [16], standards [15, 62], and better testing tools [14, 18]. While these approaches are important, since ultimately developers will be able to best make their apps accessible, current progress remains slow [42], motivating research that tries to solve the problem without relying on the original developers' cooperation. Supplanting or automatically generating metadata for existing inaccessible apps may provide more immediate benefits for users. To this end, Interaction Proxies allows missing metadata to be supplanted by end-users by manually labeling UI elements and forming a shared repository of such information for runtime repair [80, 81]. However, this approach requires active volunteers to update and maintain annotations and customizations for a large number of apps which are frequently updated. The largest prior attempt to "crowdsource" accessibility, Social Accessibility for the Web [72], showed early promise but ultimately was unable to make a big dent in the problem of Web accessibility.

Previous research in automated image description generation [35, 39] and UI label prediction [23] also creates specific types of accessibility automatically metadata from pixels. However, it only comprises a subset of the information needed for accessibility services to work as expected (i.e., these approaches do not make missing UI elements available). Our approach goes much further toward complete accessibility metadata generation from pixels, automatically inferring missing UI elements (size, position, type, content, and state), proper navigation order, and grouping.

2.2 UI Detection from Pixels

Our work involves inferring the locations and semantics of UI elements on a screen to provide metadata for accessibility services. Some early work extracted UI information to modify interfaces from non-pixel sources, such as the accessibility [17] and instrumentation APIs [32, 58], or software tutorial videos [11]. However, they require API access to enable instrumentation or to expose UI elements, and often have incomplete access to UI metadata. Pixel-based approaches, in contrast, are more independent of the underlying app toolkit and do not require UIs to be exposed via APIs.

Using pixels to identify UI elements has long been used to make apps accessible. For instance, the OutSpoken screen reader for Windows 3.1 allowed users to label icons on the screen, which it then recognizes from their pixels alone [69]. Inferring information from pixels of interfaces has been applied in diverse applications such as interface augmentation and remapping [11, 17, 30, 79], GUI testing [78], data-driven design for GUI search [20, 22, 45] or prototyping [70], generating UI code from existing apps to support app development [12, 21, 24, 53, 57], and GUI security [25]. Some work also employed pixel-based methods to improve accessibility, such as Prefab, which augments existing app interface with target-aware pointing techniques that enhance interaction for people with motor impairments [31].

There are multiple approaches to pixel-based interpretation of interfaces. Recent work by Chen et al. [24] categorizes and evaluates two major GUI detection approaches: using traditional image processing methods (e.g., edge/contour detection [57], template matching [30, 61, 78]) and using deep learning models trained on large-scale GUI data [21, 24].

Traditional image processing methods: Edge/contour detection methods [57, 70, 73] detect and merge edges into UI elements, which can work well on simple GUIs, but can fall short on images (e.g., gradient background, photos) or complex GUI layouts. Alternatively, template matching methods [30, 78, 79] may work better on these cases as there are more features in complex UI elements. However, they require feature engineering and templates to recognize the shape and type of UI elements, which may limit them to detecting UIs without much variance in visual features.

Deep learning models: Pix2Code [12] applies an end-to-end neural image captioning model to predict a Domain Specific Language description of the interface layout. Chen et al. [21] applies a similar CNN-RNN model to generate a UI skeleton (consisting of widget types and layout) from a screenshot. In both works, the generated layouts only provide relative locations of UI elements in the view hierachy, while most accessibility services require the absolute location of each UI element on the screen. Object detection models can also locate UI elements on a screen: GalleryDC [20] detects UIs with Faster RCNN model [64] to auto-create a gallery of GUI design components, and White et al. [75] applies YOLOv2 model [63] to identify GUI widgets in screenshots to improve GUI testing.

Hybrid methods: Some work [24, 26, 53] first uses traditional image processing methods (e.g., edge detection) to locate UI elements, and then applies CNN-based classification to determine the semantics of UI elements (e.g., UI type).

Many of these approaches aim at facilitating the early stages of app design rather than accessibility; therefore, they only focus on a subset of metadata (i.e., UI location and type) needed by accessibility services. In contrast, our deep learning-based approach is optimized for accessibility use cases, provides more comprehensive metadata from pixels (e.g., UI state, navigation order, grouping), and requires less resources to efficiently run on mobile phones.

2.3 Understanding UI Semantics

Beyond detecting UI elements from a screenshot, our work requires obtaining a more thorough understanding of UI semantics (e.g., UI type, state, navigation order, grouping, image and icon descriptions), which can further improve accessibility services and user experience. Beyond UI element detection, recent data-driven design work relies on large UI design datasets to infer higher level interface semantics, such as design patterns [56], flows [28], and tags [19]. Screen readers require inferring a different set of semantics, such as the navigation order and UI hierarchy (i.e., grouping) for a screen. Some work applies machine learning on UI design datasets to infer interface layout [13, 21, 22, 48]. Other work extracts interface structure through system APIs [52]. In contrast, we worked with people with visual impairments to create a heuristics-based approach to group and provide a navigation order for detected UI elements to better fit the expected user experience of a screen reader.

In addition, it is important for accessibility services to know the state and interactivity of an element (e.g., clickability, selection state), so they can handle elements differently (e.g., VoiceOver will announce "Button" for clickable UI elements; Switch Control may skip non-interactive UI elements in navigation to save time). The work by Swearngin et al. [71] predicts the human perception of the tappability of mobile app UI elements, based on human annotations and common clickability signifiers (e.g., UI type, size). In our work, we built and integrated a model to predict the clickability of some UI elements based on similar features (i.e.,, size, location, icon type); however, our goal is to predict the actual clickability of UI elements rather than the human perception; as [71] shows, these two often mismatch. Beyond previous work to detect UI elements from screenshots [12, 24, 53, 57], we additionally detect the selection state for relevant UI element types (e.g., Toggle, Checkbox) by inferring additional UI element type classes.

Lastly, we infer semantic information from UI elements to improve the screen reader experience. To recognize icons, previous work [50, 76, 77] trained image classification models from UI design datasets [27]. To describe content in pictures, prior work used deep learning models to generate natural language descriptions of images [44, 46], and some accessibility improvement research has also leveraged crowdsourcing to generate image captions [35, 39, 40]. We use an existing Icon Recognition engine and Image Descriptions feature in iOS [4] to generate alternative text for detected icons and pictures, respectively.

3 iOS APP SCREEN DATASET

To train and test our UI detection model, we first created a dataset of 77,637 screens from 4,068 iPhone apps. Creating our dataset required two steps: *(i)* collecting the screens, in which we captured screenshots and extracted information of their UI trees and accessibility trees, and *(ii)* manually annotating the visual UI elements on the screens. We fine-tuned this process, including our custom data collection and annotation software, through several pilot trials. The detail of workers we recruited in data collection and annotation and the instructions for annotators are in supplementary material.

3.1 Screen Collection

We collected a total of 80,945 unique screens from 4,239 iPhone apps, manually downloading and installing the top 200 most-downloaded free apps for each of 23 app categories (excluding Games and AR Apps) presented in the U.S. App Store. Note that some apps are in multiple categories. We skipped apps that required credit card information (e.g., for in-app payment) or sensitive personal information (e.g., log-in for banking apps, personal health data). We created a set of placeholder accounts to log into apps that required this in order to access more of their screens. We collected all screens between 12/09/2019 and 02/13/2020.

Ten workers manually traversed screens in each app to collect screen information using custom software we built. For each traversed screen, our software collected a screenshot and metadata about UI elements. The metadata includes a tree structure of all exposed UI elements on a screen, and properties of each UI element (e.g., class, bounding box, trait). We also extracted accessibility properties of each UI element that are exposed to the Accessibility APIs (e.g., isAccessibilityElement, hint, value). Our collection software also assigned a unique identifier to each UI element. The collected metadata has the same limitations in terms of completeness and correctness that motivated our approach. Therefore, establishing a reliable ground truth requires annotating the collected screens.

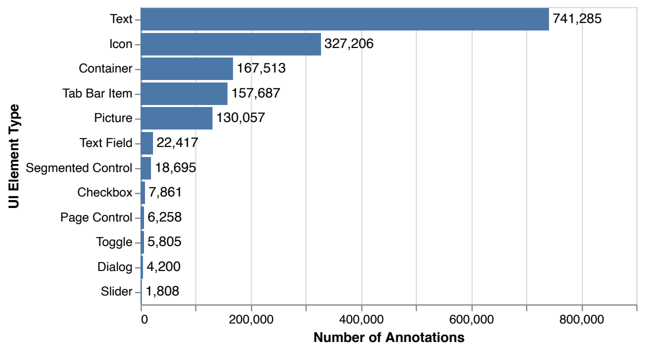


Figure 2: The number of annotations for each UI element type represented in the dataset.

3.2 Screen Annotation

Forty workers annotated all visually discernible UI elements in the screenshots. The annotation process had two components: *(i)* segmentation and *(ii)* classification.

In segmentation, workers determined a bounding box for each UI element. The annotation tool helped to direct workers toward using a bounding box provided by a captured UI element, when an appropriate one was available, to improve consistency across workers and annotations. When no bounding box was available for a UI element, the annotators manually drew a box. We established guidelines to ensure consistency and quality, such as how to draw a tight bounding box; when to group or separate bounding boxes; what elements to ignore (e.g., background); and how to handle edge cases (e.g., clipped elements).

For classification, workers assigned attributes to the identified UI elements. For each, annotators assigned one of 12 common UI types based on visual inspection: Checkbox, Container, Dialog, Icon, Picture, Page Control, Segmented Control, Slider, Text, Text Field, Tab Bar Item, and Toggle. Visual examples of UI types are available in our Supplementary Material. Two authors of this paper (a researcher and a senior accessibility engineer) chose these UI types by examining 500 samples screens to identify which elements are important for accessibility services. For UI elements not in one of these UI types, such as advertisements and map views, annotators marked them as Other (0.5% of all annotations). Annotators also labeled additional information for specific UI types, such as whether each Checkbox or Toggle is selected, whether an element is clickable, and which part of an element the bounding box encapsulates (e.g., a Text Field's outline or its placeholder text). As the attributes can sometimes be ambiguous, annotators could also choose the option "unsure".

To monitor data quality and iteratively correct operational errors, two researchers examined a random selection of 6% of the screens after each of the 20 batches was annotated. In addition, our Quality Assurance (QA) team examined 20% of screens that were annotated in each batch and shared a detailed QA report. Based on this, we worked with the annotators to reduce common errors. The total error rate we observed was 2.62% in all batches, which dropped from 4.49% (first batch) to 1.27% (last batch). On average, the error rate of bounding boxes (e.g., missing or extra annotation, too big or small of a box) was 1.35%; the error rate of UI type was 0.54%; and, the error rate of UI attributes (e.g., clickable/not clickable, selected/not selected) was 0.73%. We found that the primary errors were missing annotations for pictures and confusion between clickable and not-clickable elements, a known challenge [71].

During the annotation phase, we discarded 3,308 screens that were accidentally captured when an app was in the transition between screens, accidentally showed an AssistiveTouch button, would take too long to annotate (e.g., contain 50+ UI Elements), or had much non-English text. All of these screens represent opportunities for future work, but we considered them out of scope for the first version of the project. This resulted in 77,637 total annotated screens.

3.3 Dataset Composition Analysis

To better understand the composition of our dataset, we conducted two analyses. The first analysis explores our dataset's biases between different UI types, which may impact our model performance. The annotations revealed an imbalance of UI types in app screens, as shown in Fig. 2: Text has the highest representation (741,285 annotations); Sliders have the lowest (1,808 annotations); and the top 4 UI types comprise 95% of annotations. We consider such data imbalances in model training to improve performance on under-represented UI types, as discussed in Section 4.1.

Our second analysis examined discrepancies between annotations and UI element metadata for each screen to estimate how many UI elements were not available to accessibility services. Note that there are important limitations to this estimate, as these annotations are not a perfect reflection of all UI elements that are important to accessibility service: annotators sometimes ignore UI elements; annotations may include UI elements created for aesthetic reasons that are not important for accessibility; annotations may have additional errors as mentioned in Section 3.2. Furthermore, the available UI element metadata do not account for all possible ways to make an app accessible. For instance, some apps use custom methods (e.g., re-implement a screen reader) when their UI toolkits do not support native accessibility. Our analysis focused on UI elements exposed to the native screen reader with the understanding that annotations from human observation are still a close approximation to important accessibility elements. In most cases, the differences between annotations and existing accessibility metadata suggest potential accessibility issues in a screen.

Matching annotations to UI elements in the view hierarchy presents several challenges. Annotation bounding boxes sometimes do not line up exactly to the underlying UI element's. Sometimes they differ in the way they partition an overarching UI component (e.g., treating individual lines or paragraphs of Text as individual elements versus one large piece of Text, or exposing separate elements for the Icon, Container, and Text of a button versus just exposing one of these parts to accessibility services). Some of the larger exposed accessibility elements also have many smaller accessibility elements nested inside of them, creating further ambiguity for which elements match to which annotations.

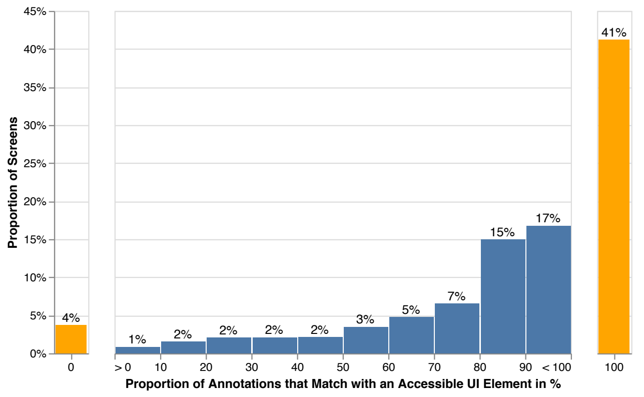


Figure 3: A histogram of the percentage of screens binned by what percentage of their annotations matched with an existing accessible UI element, with separate bars to show the percentage of screens with 0% matches and 100% matches.

To make this comparison, we ignored advertisements or maps and examined the remaining annotations with the following procedure (which we describe in more detail in this paper's supplemental material): categorize annotations by whether any bounding box B from a non-fullscreen accessibility element contains or overlaps with annotation's bounding box A, where we define "containment" as whether at least 85% of A's area lies within B, and "overlap" as whether the intersection over union of A and B is at least 5%. When an exposed element's bounding box contains only one annotation, we consider them as a match with each other. We consider annotations that have no overlap with any exposed accessibility elements to not have a match, with a few exceptions like Icons that horizontally align with already-matched Text elements that are not independently clickable, as is often the case in table cells or buttons. For the remaining possibilities — an exposed element contains more than one annotation, or overlaps but does not contain an annotation — we examined 150 example screens with different UI element types to develop heuristics, which we tested on another 150 example screens. These heuristics try to match inconsistent bounding boxes for Pictures and Text elements, nested bounding boxes, and functionally redundant annotations. To maintain a high recall on matches, these heuristics have lenient tolerances and do not require clickable annotations to match with clickable UI elements. Hence, we may have underestimated the number of un-matched annotations.

As shown in Fig. 3, 59% of screens have annotations that do not match to any accessible UI element (a mean of 5.3 and median of 2 annotations per screen). We found that 94% of the apps in our dataset have at least one such screen, rather than just a few unusable apps contributing to a large number of these screens. In these screens, our accessibility metadata generation approach has the potential to expose missing accessibility elements. Our approach could be especially useful in providing accessibility metadata for the 4% of screens that do not expose any accessibility elements. As shown in Fig. 4, the prevalence of matches between annotations and accessible UI elements varied by UI type. Of all unmatched annotations, 33% are Text and 21% are Picture. The discrepancy

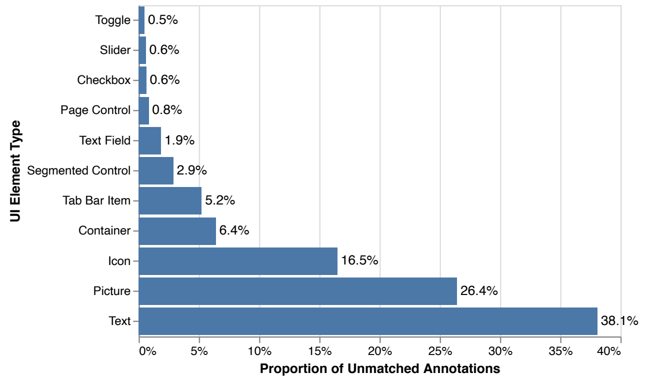


Figure 4: A bar chart showing the breakdown of unmatched annotations by UI Element Type.

between annotations and existing accessibility metadata demonstrates the importance of annotation, which provides additional information compared to prior mobile UI collection datasets [27].

4 UI DETECTION MODEL

We trained an object detection model to extract UI elements from pixels, which locates UI elements on an app screenshot and predicts their UI types. After experimenting with a variety of network architectures and configurations, we chose one that balances accuracy and on-device performance. We describe the model architecture, data, and evaluation below.

4.1 Model Architecture

We started by experimenting with Faster R-CNN (and its extension Mask R-CNN) [43, 64] which is one of the best-performance object detection models evaluated on public datasets. As the goal is to run this detection model on-device (iPhone), the inference time and memory footprint became limiting factors. Faster R-CNN takes more than 1 second for each screen and more than 120 MB memory, unsuitable for on-device inference. We then experimented with TuriCreate Object Detection toolkits [6], which is easy to use and optimized for iOS devices. The TuriCreate model uses approximately half of the memory (60M) and has a significantly faster inference time (around 20ms). In the pursuit of a more efficient model with tighter bounding box predictions and a higher mAP (mean Average Precision), we converged on a SSD (Single Shot MultiBox Detector) model [51] that meets our requirements. Specifically, we used MobileNetV1 (instead of large ResNet) as a backbone to reduce memory usage. Another common challenge in object detection tasks is detecting small objects. Unfortunately, UI elements are relatively small compared to most targets often seen in object detection tasks. We used Feature Pyramid Network (FPN), which is a feature extractor designed with a hierarchical pyramid to improve accuracy and speed when detecting objects in different scales. To handle class-imbalanced data, discussed in Section 3.3, we performed data augmentation on underrepresented UI types during training, and applied a class-balanced loss function (more weight for underrepresented UI types). Our final architecture uses only 20MB of memory (CoreML model), and takes only about 10ms for inference (on an iPhone 11 running iOS 14). To train this model, we used 4 Tesla V100 GPUs for 20 hours (557,000 iterations).

Object detection models often return multiple bounding boxes for a given underlying target with different confidences. We use two post-processing methods to filter out duplicate detections: (i) Non-Max Suppression picks the most confident detection from overlapping detections [55]. (ii) Different confidence thresholds can be applied on each UI type to remove less certain detections. We picked confidence thresholds per UI type that balance recall and precision for each (Figure 5).

4.2 Training and Testing Data

The training and testing datasets include 13 classes (Table 1) derived from the UI types in annotation: Checkbox (Selected), Checkbox (Unselected), Container, Dialog, Icon, Picture, Page Control, Segmented Control, Slider, Text, Text Field, Toggle (Selected), and Toggle (Unselected). Based on initial experiments with UI detection models, we split the Checkbox and Toggle into "Selected" and "Unselected" subtypes, and remapped the Tab Bar Item into Icon and Text.

Checkbox and Toggle: The visual appearance of "Selected" and "Unselected" Checkboxes are quite different (same for Toggles), and splitting them improved the overall detection accuracy for each class. Directly detecting selection state also means we do not need a separate classification model to determine the selection state of a detected Checkbox or Toggle, which reduces on-device memory usage and inference time.

Tab Bar Item: Initial experiments revealed that the standalone Tab Bar Item class had good recall but low precision. This outcome was because our initial model tended to detect most UI elements at the bottom of screen as Tab Bar Items, including Icons and Text at the bottom of screen of apps that did not have Tab Bars at all. Therefore, we decided to split the Tab Bar Item annotations into Text and/or Icon classes. Later, we will describe heuristics that we introduced to group rows of Text and Icons near the bottom of the screen into Tab Bar Items contained within a Tab Bar.

Regarding our training/testing data split, a simple random-split cannot evaluate the model generalizability, as screens in the same app may have very similar visual appearances. To avoid this data leakage problem [47], we split the screens in the dataset by app. We also ensured the representation of app categories, screen count per app, and UI types are similar in the training and testing datasets. We ran several random splits on apps, and stopped when the split satisfied these requirements. The resulting split has 72,635 screens in the training dataset and 5,002 screens in the testing dataset.

4.3 Evaluation

For each UI type, we evaluated our model performance using Average Precision (AP), a standard object detection evaluation metric. We choose a threshold of > 0.5 IoU (Intersection over Union), commonly used in object detection challenges [33], to match a detection with a ground truth UI element. On our testing dataset (5,002 screenshots), our UI detection model achieved 71.3% mean AP. If we weigh our results by the frequency of each UI type in the dataset, the weighted mean AP is 82.7%. Table 1 shows the individual AP for each of the 13 UI types. Based on the precision-recall

Table 1: Average Precision (AP) of Each UI Type in 5,002 Testing Screens.

UI Type	AP (>0.5 IOU)	AP (Center)	Count
Checkbox (Unselected)	77.5%	79.1%	471
Checkbox (Selected)	27.4%	27.4%	119
Container	82.1%	83.0%	11528
Dialog	62.9%	63.3%	264
Icon	79.7%	88.0%	21875
Page Control	65.4%	87.3%	454
Picture	72.0%	76.9%	9211
Segmented Control	55.1%	57.6%	1036
Slider	55.6%	63.0%	110
Text	87.5%	91.7%	47045
Text Field	79.2%	79.5%	1379
Toggle (Unselected)	91.7%	91.7%	247
Toggle (Selected)	90.5%	92.0%	131
Mean	**71.3%**	**75.4%**	
Weighted Mean	**87.5%**	**83.2%**	

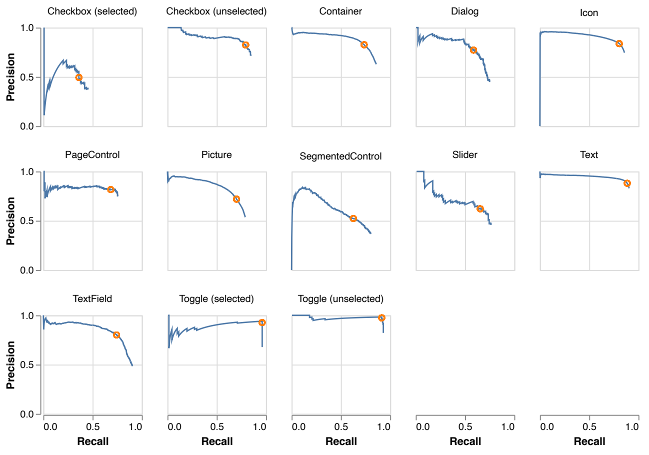


Figure 5: Our model's precision-recall curves for detecting each UI type. Examining individual performance enables us to assign a confidence threshold to each UI type, balancing recall and precision (shown as orange points).

curves (PR curves) for individual UI types (as shown in Fig. 5), we chose a different confidence threshold for each UI type. The use case of screen reader accessibility informed our choices. Note that the choices may differ for other accessibility services (e.g., Switch-Control users may prefer higher precision over recall to reduce the number of elements to navigate when they can see elements that our model fails to detect).

Our model achieved the lowest AP for the Checkbox (Selected). Its low frequency in the dataset may contribute to poor model performance; however, the Slider and Toggle also have a similar low frequency. To further understand how much the model misclassifies one UI type as another, we generated a confusion matrix [68] as shown in Fig. 6. We found that the model often confuses Checkbox

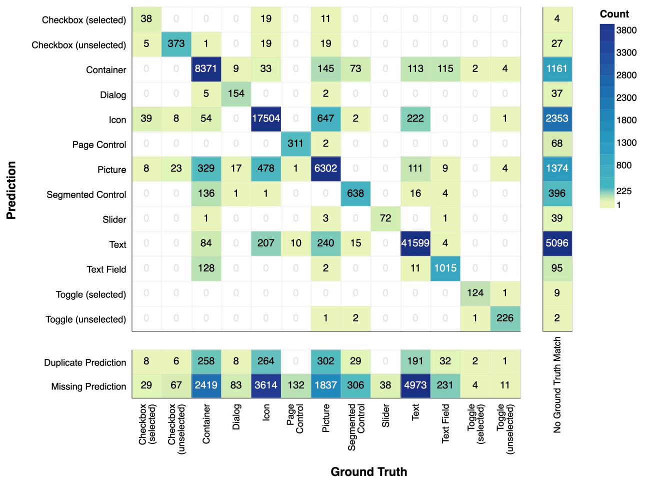


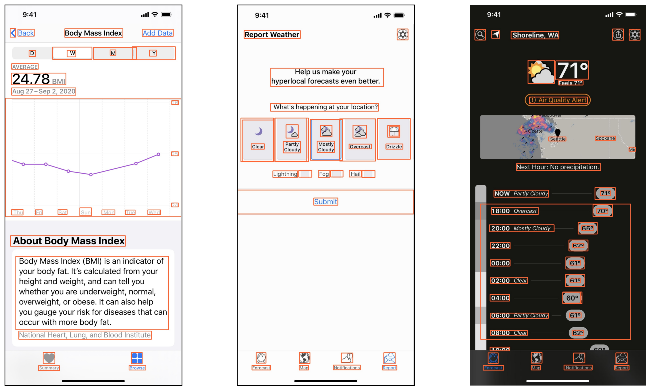
Figure 6: Confusion matrix for all UI types. Checkboxes (selected) were often confused with Icons or Pictures. Icons were sometimes confused with Pictures.

(Selected) with Icon and sometimes Picture. Checkboxes look visually similar to Icons, potentially explaining why our model tends to mis-classify them as Icons, which have much higher frequency in the dataset. Creating another classification model to distinguish Checkbox (Selected) from Icon detections has the potential to double its recall.

We also evaluated our model's detection performance with an evaluation metric specific for accessibility services: whether the center of a detection lies within its target UI element. With mobile screen readers, double-tapping a detection will pass a tap event to its center, which activates the target. This evaluation metric is more relaxed than > 0.5 IoU and increases the mean AP from 71.3% to 75.4%, as shown in column AP (Center) of Table 1. In some cases, the detection bounding box may not be tight enough to include all semantic information of a target UI element, but still enables users to manipulate the target. Thus, this metric may better capture whether the model enables people to use the manipulable UI elements.

5 IMPROVING THE USER EXPERIENCE FROM UI DETECTION RESULTS

Our model correctly detects and classifies most UI elements on each screen; however, it may still miss some UI elements and generate extra detections. Even when the detections are perfect, simply presenting them to screen readers will not provide an ideal user experience as the model does not provide comprehensive accessibility metadata (e.g., UI content, state, clickability). We worked with 3 blind QA engineers and 2 senior accessibility engineers in our company to iteratively uncover and design the following 6 improvements to provide a better screen reader experience. Over 5 months, the participating engineers tried new versions of the model on apps of their choice, and provided feedback on when it worked well and how it could be improved.


Back
Body Mass Index
Add Data
24.78 BMI
About Body Mass Index
Report Weather
Help us make your hyperlocal forecasts even better.
What's happening at your location?
Submit
Shoreline, WA
71°
Next Hour: No precipitation.

Body Mass Index (BMI) is an indicator of your body fat. It's calculated from your height and weight, and can tell you whether you are underweight, normal, overweight, or obese. It can also help you gauge your risk for diseases that can occur with more body fat.

National Heart, Lung, and Blood Institute

Figure 7: The raw UI detection results from our model would not always result in the best screen reader user experience. (Left) The model does not detect the leftmost Segmented Control. (Middle) The rightmost umbrella has two detections; Checkbox and Text are not grouped. (Right) The big Container detection at the bottom half may be removed. In all examples, Icon and Text on the tab bar should be grouped as Tab Buttons. The example screens are from Apple Health and Dark Sky.

5.1 Finding Missing UI Elements and Removing Extra Detections

Our UI detection model may miss some UI elements. As seen in Fig. 7 Left, the model sometimes only detects a subset of Segmented Controls (SC). In 99.1% of the screens in our dataset, every Text on the same row of SCs is contained by a SC. Based on this pattern, after we detect SCs in a screen, we find Text detections on the same row of detected SCs but not contained by any SC. Our heuristics update these Text elements as Segmented Controls. Another common missing UI element is Text, especially when it is small and/or on a complex background. To address this problem, we ran the iOS built-in OCR engine [5], which is trained specifically to detect text and outperforms our model on this class. We include text detections from OCR when our model misses them (i.e., OCR text has no overlap with existing detections).

Our UI detection model may also detect extra UI elements. For a UI element, our model sometimes generates multiple bounding boxes, each with a different but visually similar UI type (i.e., Picture/Icon, Segmented Control/TextField/Container). For example, in Fig. 7 (Middle), the rightmost "umbrella" has a box detected as Icon and another box detected as Picture. Therefore, we apply a customized Non-Max Suppression algorithm to remove duplicate detections within these visually similar UI types. There are also extra UI elements we should not remove (e.g., a large Picture can contain a small Icon, a Container may contain a TextField), thus our algorithm only removes spatially similar detections when IoU > 0.8.

5.2 Recognizing UI Content

UI types Text, Icon, and Picture (comprising 83.3% of UI elements in our dataset) contain rich information. Therefore, it is important to recognize and describe their contents to screen reader users. We

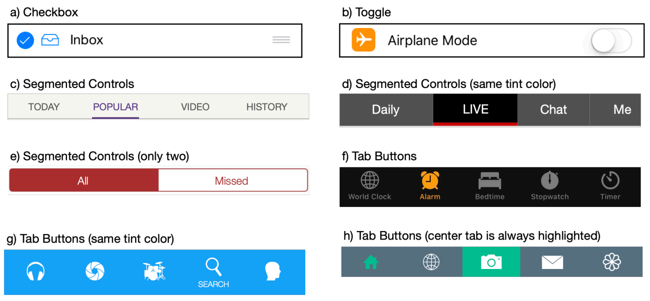


Figure 8: Examples of how selection state is visually indicated in different UI types. Differences in tint color is often a signal of selection for Segmented Controls and Tab Buttons. There are also other visual indicators for selection states.

leverage some iOS built-in features to recognize content in these UI detections. **For Text:** we use the iOS built-in OCR engine [5], which provides a tight bounding box and accurate text result with a reasonable latency (< 0.5s). **For Icon:** we leverage the Icon Recognition engine in iOS 13 (and later) VoiceOver to classify 38 common icon types [10]. **For Picture:** we leverage the Image Descriptions feature in iOS 14's VoiceOver to generate full-sentence alternative text [4].

5.3 Determining UI Selection State

Several UI types also have selection states, such as Toggle, Checkbox, Segmented Control, and Tab Button. With this metadata, VoiceOver can announce UIs with selection states (e.g., "Airplane Mode, Off", "Alarm, Tab, Selected" in Fig. 8). For Toggle and Checkbox, our UI detection model includes selection states in detected UI types. For Segmented Control and Tab Button, we leverage their visual information to determine the selection states. The most common visual indicator is the tint color. As shown in Fig. 8.f, the selected Tab has tint color while other Tabs are not highlighted. Within each Tab, we extract the most frequent color as the background color, and the second most frequent color as the tint color. Among all Tab Buttons, we find the one with an outlier tint color [60]. Finally, we assign a "selected" state to that detection, and a "not selected" state to the remaining detections. Segmented Controls also frequently use tint color as a visual indicator. As shown in Fig. 8, custom UI designs may keep the same tint color and use other visual indicators (Fig. 8.c); therefore, we tried to address some common designs with additional heuristics, such as checking the bar at the bottom of the Segmented Control (Fig. 8.d), or finding the only Tab with text (Fig. 8.g). We evaluate our selection state heuristics on our testing dataset (5002 screens). On the 936 screens in which our approach detects Tab Buttons, we assign the correct selection state 90.5% of the time. Among 337 screens in which our approach detects Segmented Controls, we assign the correct selection state on 73.6% of screens.

5.4 Determining UI Interactivity

Knowing whether a UI element is interactive is important for screen reader users. With the correct accessibility metadata, a screen reader can announce actions supported by each UI element (e.g., "Page

X of Y, swipe up or down to adjust the value" for a page control; "Password, double tap to edit" for a text field; "Menu, button" for a clickable UI element like a button, web link, table cell, etc). Several UI types detected by our model (e.g., Text Field, Page Control, Slider) already indicate their supported actions. On the other hand, Text, Picture, and Icon can be non-clickable in some cases (e.g., Airplane Icon, "Inbox" Text in Fig. 8), but clickable in other cases (e.g., Back Icon, "Add Data" Text in Fig. 7). According to feedback from screen reader users, they can often infer the clickability of Text and Picture elements from alternative text (generated by OCR and Image Descriptions), but have a difficult time telling if an Icon is clickable from the Icon Recognition result (in addition, it is unavailable when the Icon is not in the 38 supported icon types). To predict Icon clickability, we trained a Gradient Boosted Regression Trees model [3] with several features (i.e., location, size, Icon Recognition result). This model is trained with TuriCreate Boosted Trees Classifier toolkits [3]; it takes less than 10ms for inference and uses 10MB of memory (CoreML model). We only mark an Icon as clickable when our model is very confident, as screen reader users prefer higher precision over recall: if a clickable Icon is predicted as non-clickable, the user may still try to activate it; if a non-clickable Icon is predicted as clickable, the user will hear "Button" in screen reader announcements, and will be confused if the activation does not do anything. Therefore, we pick a threshold for our model to keep 90.0% precision, which allows 73.6% recall. Additionally, there are more UI interactivities worth investigating in future work, such as detecting the enabled/disabled state of UI elements.

5.5 Grouping Elements for Efficient Navigation

Grouping related UI elements makes navigation more efficient for screen readers and other accessibility services [8, 36]. Fig. 9 shows the original detections from our UI element detection model; navigating through a large amount of detections would take a long time for screen reader users to understand the screen contents. To reduce the number of UI elements in navigation, we group related elements together. For grouped UI elements, we provide access to clickable sub-elements through custom actions (supported by mobile screen readers). The user can also read each non-clickable sub-element by adjusting the speech granularity in the rotor [1]. This approach still has limitations: when our model makes mistakes in inferring clickability, users may lose access to some clickable sub-elements. However, screen reader users noted that the increased efficiency from grouping was worth the tradeoffs of losing access to some elements.

We developed multiple heuristics that group UI detections based on their UI types, sizes, and spatial relationships on the screen. Fig. 9 displays the results of our grouping heuristics on example screens. We provide the concatenated text of sub-elements as alternative text for each grouped element.

- *Text Grouping* (Fig. 9.B): We group a Text T_1 with a Text below T_2 if they satisfy: 1) they have x-overlap, and 2) the y-distance between the two texts should be less than a threshold – we choose $min(T_1.height, T_2.height)$. We do this repetitively to group multi-line text.
- *Tab Grouping* (Fig. 9.B): We group Icon and Text detections into Tab Buttons by 1) finding the bottom-most detection B; 2) determining if a Text or Icon detection D is in the tab bar if D is in the bottom 20% of the screen, and $B.bottom$ - $D.top$ is within a threshold (0.08 * screen height) so that D is not much higher than B; 3) grouping Icon and Text detections that have x-overlap together, and determined to be in the tab bar, as one Tab Button.
- *Container Grouping* (Fig. 9.D): We group a Container detection with all detections inside it as one element.
- *Picture Subtitle Grouping* (Fig. 9.D): We group Text T_1 as a descriptive subtitle with Picture P if they satisfy: 1) they have 50%+ x-overlap with P, 2) T_1 has a y-distance to P less than a threshold (we chose 0.03 * screen height), and 3) T_1 is closer to P than to any other detection below T_1. We additionally group a second line of Text T_2 below T_1 if it satisfies the same conditions in relation to T_1.

To evaluate these heuristics, we ran our grouping heuristics on a subset of collected screens (n=300, randomly picked). Overall, our grouping heuristics reduced the number of UI elements to navigate by 48.5% (Without Grouping - Mean: 21.83, Std: 12.8, With Grouping - Mean: 12.1, Std: 7.6). Two researchers (authors of this paper) manually tabulated grouping errors across these screens using a rubric which contains 4 grouping heuristic categories (i.e., text grouping, container, picture subtitle, tab), and "Other" for additional errors. Grouping errors are classified as "Should have grouped" if two or more elements should have been grouped together, or "Incorrectly grouped" if two or more unrelated elements were grouped. As there may be multiple groupings for some elements, we did not count a grouping or missed grouping as an error if it was ambiguous. We acknowledge that human observation is not perfect, and there might be bias in doing evaluation by the authors. To mitigate these concerns, the researchers first independently evaluated 10% of screens and then discussed the errors found until reaching a sufficient level of agreement. After reaching consensus, each researcher examined half of the screens, and found very similar numbers of grouping errors across independent sets of screens. Among 1568 groups made, 88 (5.6%) were "Incorrectly grouped". In addition, we found 99 "Should have grouped" errors. There were 0.62 errors per screen (Min=0, Max=8, Std=1.1).

The largest set of grouping errors were from our *text grouping* heuristic (25.7%), followed by *picture subtitle* (16%), *tab button* (8.6%) and *container* (7.5%). 42% of the grouping errors did not fall into one of our heuristic categories: the majority of errors (48 errors) consisted of missed grouping text with contained icons, remaining OCR-detected text inside icons, and grouping of unrelated text. While evaluating grouping errors, we also found opportunities to further improve the user experience (e.g., group text and text field below it), which may lead to new heuristics.

While our heuristics in general performed well, we note that these heuristics are likely to be more successful when applied to iOS app screens due to the influence of native user interface widgets and design guidelines (i.e., Human Interface Guidelines). For example, our Tab Grouping heuristic likely works well because iOS provides a standard widget, which a significant amount of apps use, to display tabs at the bottom of an app screen (i.e., UITabBarController). As such, our heuristics may need to be adapted for other platforms

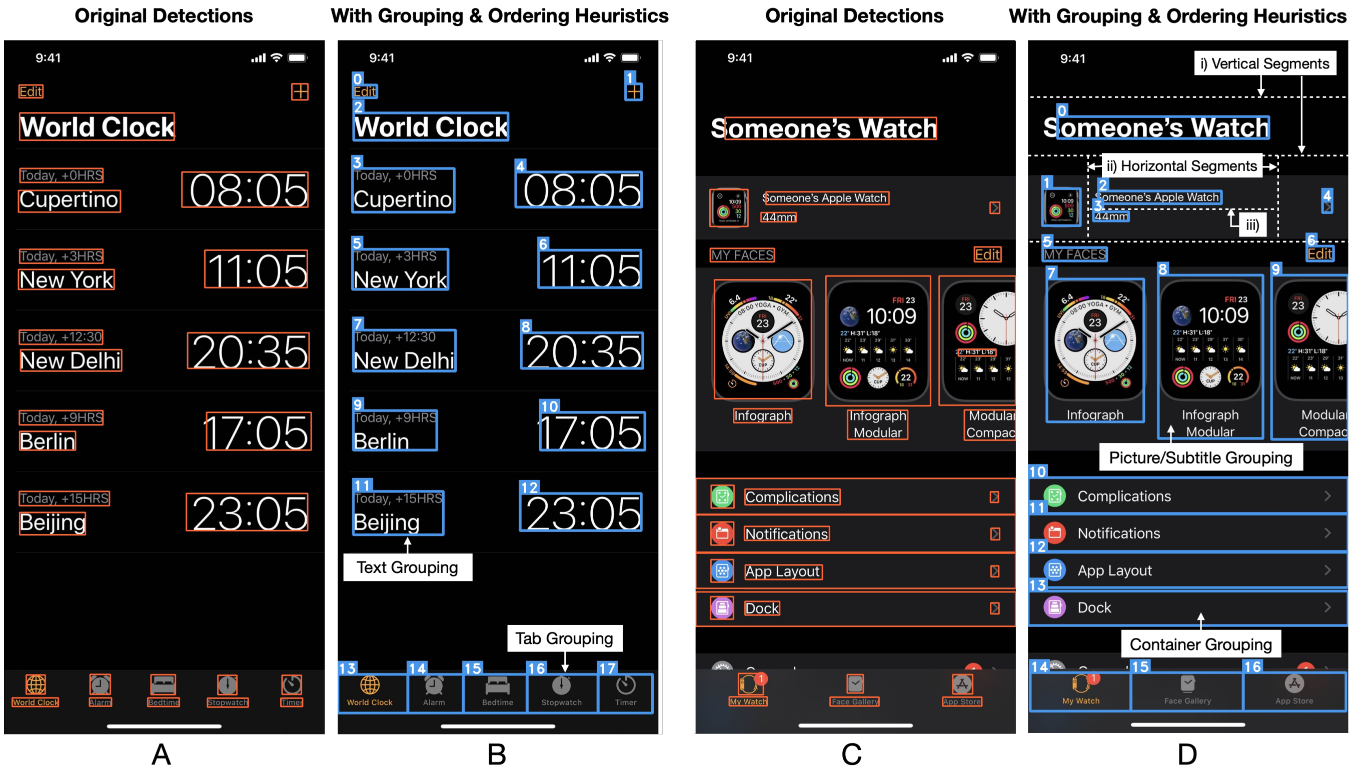


Figure 9: Example screens to demonstrate our grouping and ordering heuristics: *(A, C)* UI detections from our model in screens of Apple Clock and Apple Watch apps, *(B)* the output demonstrating text and tab button grouping, and *(D)* the output demonstrating picture/subtitle and container grouping. Our ordering heuristics infer the navigation order (shown on the top-left of each box), which recursively divides the screen into vertical and horizontal regions (visualized examples with dashed line).

(e.g., Android) if the platform's widget designs are quite different from iOS designs.

5.6 Inferring Navigation Order

Mobile screen readers allow swiping to navigate all UI elements on a screen in a logical order. Therefore, we need to provide our detected UI elements to screen readers in a reasonable navigation order. After applying our grouping heuristics, we determine the navigation order with a widely used OCR XY-cut page segmentation algorithm [41, 54], which sorts text/picture blocks of a document in human reading order. We apply the algorithm to our problem, as demonstrated in Fig. 9.D: (i) We build vertical segments by determining Y coordinates where we can draw a horizontal line across the screen without overlapping any element. (ii) Within each vertical segment, we build horizontal segments by locating X coordinates where we can draw a vertical line down the segment without overlapping any element. (iii) We recursively call this algorithm on all horizontal segments to further create vertical segments and so forth. When we can no longer subdivide a segment, we order the elements inside by top-to-bottom. If there is a tie, we order them left-to-right. Within grouped elements, we also apply the XY-cut algorithm to order sub-elements. Fig. 9 shows a number on the top-left of each element to indicate the navigation order we provide to screen readers. In Fig. 9.B, a screen reader user would hear "Edit", "Add", "World Clock", and "Today, plus 0 HRS, Cupertino" for the first four elements.

To evaluate our ordering heuristics, we ran them on a subset of collected screens (n=380, randomly picked). Ten accessibility experts in our company annotated the ground truth navigation order. For each screen, we compared our heuristics output and the ground truth navigation order with a metric often seen in list sorting: the minimum insertion steps to make our output the same as ground truth (in one step, an element can be inserted to any other position in list). Our heuristics output perfectly matched the ground truth for 280 screens (73.7%), while 345 screens (90.8%) had less than 1 error (i.e., 1 insertion step) for every 10 elements. On average, there were 18.7 elements per screen (min=1, max=92, std=13.8); our heuristics produced a mean of 0.67 errors per screen (min=0, max=17, std=1.79).

To understand where our ordering heuristics could be improved, we examined 35 screens with more than 1 error for every 10 elements. On 12 screens, we found the ordering was ambiguous (i.e., there were multiple reasonable orderings). For the remaining screens, 19 of them would have a better navigation order by improving our grouping heuristics: (i) our heuristics grouped some text with pictures incorrectly, or our *picture subtitle* grouping heuristic did not group some text as a subtitle which caused ordering errors

(13 screens); (ii) our *text grouping* heuristics grouped some text incorrectly or should have grouped some text which caused ordering errors (9 screens); (iii) our heuristics detected some icons inside of text blocks but did not group them with the text, so the icons were ordered after the text (6 screens).

6 USER STUDIES

To understand how our approach impacts the user experience of screen readers, we conducted a user study with 9 people with visual impairments (3 Female, 6 Male). They are all regular iOS VoiceOver users with an average of 8.9 years (min = 6, max = 11) of VoiceOver experience. The average age of participants was 44.2 years (min = 28, max = 65). In the studies, we asked participants to use apps with regular VoiceOver and Screen Recognition (our approach) and compare their experiences. There were two types of studies: *(i)* Interview studies: through video conferencing with 4 screen reader users (1F, 3M) who are accessibility QA engineers in our company, and *(ii)* Email studies: a remote unsupervised usability study through emails with 5 screen reader users (2F, 3M) who volunteered to participate, which provides more time flexibility than interviews. To reduce bias, we excluded the QA engineers who provided suggestions in Section 5 from our studies.

6.1 Procedure

Both studies used the same instructions and questions. We asked participants to select 3 iOS apps that they found difficult or impossible to use with VoiceOver. If participants could not think of such apps, we offered a list of suggestions. We asked the participants to spend 10 minutes using each app, first with regular VoiceOver and then with Screen Recognition. During the study, participants were always aware of the current condition they were testing. We did not counterbalance the order of conditions, as all participants were experienced VoiceOver users and most apps in the study were suggested by participants (they already tried these apps with regular VoiceOver before). We collected feedback from participants about the differences they experienced between regular VoiceOver and Screen Recognition, benefits from Screen Recognition, and challenges encountered using Screen Recognition.

Due to the COVID-19 pandemic, we conducted all user studies remotely. In the Interview studies, we sent study preparation through email in advance and feedback was collected through a conference call. In the Email studies, we sent and collected all study instructions and feedback via email.

6.2 Results

Every participant chose apps to use during the study that they had previously found difficult to use with VoiceOver. Participants used 22 different apps, only 2 of which came from our suggested list. Only 2 apps were explored by multiple participants: one from our suggested list (chosen by 4 participants), and one not on the list (chosen by 2 participants). Note that we do not share specific app names because we want to be respectful and not single out individual developers. Participants chose a mix of apps with varying usability levels for VoiceOver and rated them from 1 (not usable at all) to 5 (completely usable). Most participants chose 3 apps that they already knew to be inaccessible, but 3 participants also decided

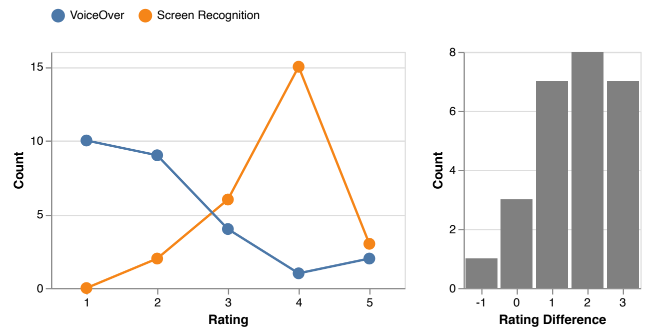


Figure 10: (Left) A histogram of usability ratings for apps using regular VoiceOver and Screen Recognition. (Right) A histogram of the rating differences within individual apps between the two systems, as the rating for Screen Recognition minus the rating for regular VoiceOver.

to include an app that they knew to be accessible which provided useful insights. All apps that participants chose were available on the iOS App Store as of August 2020. Some apps were built by major tech companies, but most were made by smaller developers or non-tech companies, such as media companies.

Overall, usability ratings for apps using regular VoiceOver averaged 2.08 (SD 1.20), whereas ratings for apps using Screen Recognition averaged 3.73 (SD 0.78). This difference is significant according to a Wilcoxon signed-rank test ($p < 0.00004$). Figure 10 shows a histogram of usability ratings, and a histogram of rating differences within individual apps between the two systems. It was clear to participants that the researchers created Screen Recognition, so these results may reflect some bias [29, 74], but still show quite a meaningful difference between the two systems, especially on the inaccessible apps that participants picked. In 4 cases, participants gave Screen Recognition an identical or worse usability rating than regular VoiceOver. Two of these cases were for apps that the participant rated very usable (5) with regular VoiceOver, and then still rated Screen Recognition as usable with either a 4 or 5. In the other two cases, both experiences received the same low rating (2 and 3 respectively). Based on participants' feedback and our own use, we believe these latter ratings reflect that these apps are difficult to use in general, even for sighted individuals.

For apps that participants rated unusable with regular VoiceOver, all noted that these apps exposed user interface elements without labels, and that it was frustrating to know that a control existed but not know its purpose. When exploring the same apps with Screen Recognition, all participants remarked that they became aware of user interface elements that they did not realize existed because they were not exposed in the regular VoiceOver interface. One participant noted that in one app a slider became available, which they found surprising because these elements are often unavailable or unusable. In another case, a participant said:

> QA2: *"I am so excited I just NEED to share. Last night I downloaded [app name removed]...which has no AX elements and is completely unusable. So I decided to enable [Screen Recognition]. Guess who has a new high score? I am in AWE! This is incredible. It is the first*

mainstream game I've ever been able to play on iOS besides Trivia Crack."

Three participants chose to try apps that they knew to be accessible. In these cases, the participants still compared Screen Recognition favorably to regular VoiceOver, remarking that Screen Recognition exposed certain features in a different and potentially useful way. For example, one participant noted that the screen layout was revealed to them in a way that they often missed, saying:

P1: *"It was fun to discover the contents of the profile pictures displayed next to the person name and understand the layout of the app. Obviously, when making the app accessible, we oftentimes lose the understanding of the screen layout. With [Screen Recognition] we now have the ability to understand the app spatially in addition to contextually."*

The only Screen Recognition experience rated as less usable than the regular VoiceOver experience occurred with a social media app. The participant that introduced this app noted that it was very accessible and gave it a usability score of 5 with regular VoiceOver, whereas they gave Screen Recognition a usability score of 4. The primary reason for this difference was the grouping choices from app developers, which made navigating through many tweets quick and focused on only the essential content. The participant noted that navigating this interface with Screen Recognition was much slower, which includes a variety of low priority content (e.g., number of likes, number of retweets, hash tags, etc.) that the app interface with regular VoiceOver omitted. However, this participant noted that they saw value in the Screen Recognition interface in certain situations, because, although slow, it did allow access to certain content that wasn't available otherwise. Being able to switch between the two modes enabled this participant to drill down into the interface content when necessary, while also enjoy a streamlined experience the rest of the time.

This example illustrates an important result: Even with Screen Recognition, the most accessible experiences are still created by app developers who understand how the content can best be conveyed to meet users' needs. Our system may not have the higher-level understanding of app interfaces to realize that the tweet list should be streamlined, while app developers can better understand the design and implement accessibility experience for users.

Screen Recognition has room for improvement. Two participants noted that while many controls became apparent with Screen Recognition, the labels on these controls sometimes required interpretation. For example, an icon labeled "down arrow" might cause an item to move down in a list, but did not expose this higher level semantic. In these cases, the participants remarked that they were able to figure out the user interface with some experimentation. Another limitation of Screen Recognition is that it cannot expose features that do not have a clear visual affordance. For example, a participant discovered that with Screen Recognition, it was impossible to turn the pages in a book reading app, even though Screen Recognition made the previously unreadable book text available. In this app, the page turning was actuated by a swipe from the screen edge, which had no visual affordance for Screen Recognition to recognize. A challenging problem for future work could be to better understand the semantics of the visual interface. Finally, one participant noted that they occasionally experienced lag with Screen Recognition on especially complex screens, which helps justify our intentions to later explore optimizations for our system.

7 DISCUSSION & FUTURE WORK

Despite persistent effort in developer tools, education, and policy, accessibility remains a challenge across platforms because many apps are not created with sufficient semantic metadata to allow accessibility services to work as expected. In this paper, we introduced an approach that instead generates accessibility metadata from the pixels of the visual interfaces, which opens a number of opportunities for future work.

A number of next steps follow directly from the approach and results presented in this paper. A clear next step is to keep improving our model and our heuristics to make them more accurate. Our current approach generates metadata from scratch over and over again; future work may explore how to allow this metadata to persist across invocations of an app or even across devices. Another next step is to apply our approach to other mobile platforms (e.g., Android) and also beyond the mobile context, such as for desktop or Web interfaces, which are similarly inaccessible due to a lack of appropriate semantics provided by developers. Applying our approach to the desktop or Web would introduce new challenges, as our current method takes advantage of the size and complexity constraints of mobile apps.

In the work presented here, we used only pixel information to create the semantic metadata needed for accessibility services. We did not leverage the semantic data inside the app view hierarchy, which accessibility services use [10, 38]. While combining information from our approach and view hierarchy could reconstruct a more complete set of accessibility metadata, we found that implementing this idea presents engineering challenges for effectively extracting and merging the two representations on-the-fly. Currently, users must enter a separate mode to access the automatic metadata; as we explore combined modes, we suspect we will need to even more carefully consider how to convey the inferred metadata to users. Furthermore, questions remain about how to resolve incompatible differences between the representations, especially in regard to higher-level concepts, such as the grouping and ordering. Instead of post hoc merging, an interesting opportunity for future work could be to build a model that leverages the structured information in the view hierarchy in addition to the pixels.

Our model and heuristics have multiple other use cases to explore. We demonstrated how we can apply them to improve accessibility on users' devices in real time. Another opportunity is to incorporate the model into developer tools to help developers create more accessible apps from the start. Current accessibility evaluation tools mainly depend on the (potentially incomplete) metadata developers already provide, limiting the tools' access to information to inform their accessibility suggestions. Our approach for understanding UIs from pixels could enable developer tools to identify more accessibility issues and automatically suggest fixes. Finally, while we have focused on accessibility use cases, the accessibility APIs also form the foundation of most on-device automation [49], and so we expect our approach to find utility in many other use cases.

The longevity of our model is uncertain. Visual app designs change over time, both in terms of aesthetic qualities and in relation to their target devices. While people are generally able to adapt to such visual changes, our model may need to re-trained. It would be ideal to create a more universal model that is resilient to such changes or could even update itself while visiting new apps. While the uncertainty remains, our dataset, as we update it over time, will provide a window into how these visual changes occur and how they affect automatic methods operating on them.

Finally, our approach could be seen as reactive to the way most app development currently prioritizes visual design, instead of approaching the problem from an accessibility first perspective. We hope that by automating some aspects of accessibility metadata generation, especially through future enhancements to developer tools, our approach could help scale basic accessibility, freeing researchers, developers, and accessibility professionals from playing catch up, and instead enable them to work toward more innovative and comprehensive accessible experiences for all.

8 CONCLUSION

We have presented an approach to automatically create accessibility metadata for mobile apps from their pixels. Our technical evaluation and user evaluation demonstrate that this approach is promising, and can often make inaccessible apps more accessible. Our work illustrates a new approach for solving a long-standing accessibility problem, which has implications across a number of different platforms and services. Going forward, we would like to use the auto-generated accessibility metadata to not only directly impact accessibility features, but also help developers make their apps more accessible from the start.

ACKNOWLEDGMENTS

We thank our colleagues at Apple, in particular: Cynthia L. Bennett, Shantanu Chhabra, Dominik Moritz, Muriel Nahmani, and Jay Pillai for their contributions.

REFERENCES

[1] Apple. 2020. About the VoiceOver rotor on iPhone, iPad, and iPod touch. https://support.apple.com/en-us/HT204783/

[2] Apple. 2020. Accessibility for Developers. https://developer.apple.com/accessibility/.

[3] Apple. 2020. Boosted Trees Classifier. https://apple.github.io/turicreate/docs/userguide/supervised-learning/boosted_trees_classifier.html

[4] Apple. 2020. iOS 14 Preview - Features. https://www.apple.com/ios/ios-14-preview/features/.

[5] Apple. 2020. Recognizing Text in Images. https://developer.apple.com/documentation/vision/recognizing_text_in_images/

[6] Apple. 2020. Turi Create. https://github.com/apple/turicreate/

[7] Apple. 2020. UIAccessibility. https://developer.apple.com/documentation/objectivec/vision/nsobject/uiaccessibility/

[8] Apple. 2020. Use Switch Control to navigate your iPhone, iPad, or iPod touch. https://support.apple.com/en-us/HT201370.

[9] Apple. 2020. Use Voice Control on your iPhone, iPad, or iPod touch. https://support.apple.com/en-us/HT210417.

[10] Apple. 2020. Vision Accessibility - iPhone. https://www.apple.com/accessibility/iphone/vision/

[11] Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. 2012. Waken: Reverse Engineering Usage Information and Interface Structure from Software Videos. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 83–92.

[12] Tony Beltramelli. 2018. Pix2Code: Generating Code from a Graphical User Interface Screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (Paris, France) *(EICS '18)*. ACM, New York, NY, USA, Article 3, 6 pages.

[13] Pavol Bielik, Marc Fischer, and Martin Vechev. 2018. Robust relational layout synthesis from examples for Android. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[14] Giorgio Brajnik. 2008. A comparative test of web accessibility evaluation methods. In *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*. 113–120.

[15] Ben Caldwell, Michael Cooper, Loretta Guarino Reid, Gregg Vanderheiden, Wendy Chisholm, John Slatin, and Jason White. 2008. Web content accessibility guidelines (WCAG) 2.0. *WWW Consortium (W3C)* (2008).

[16] Jim A Carter and David W Fourney. 2007. Techniques to assist in developing accessibility engineers. In *Proceedings of the 9th International ACM SIGACCESS Conference on Computers and accessibility*. 123–130.

[17] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. 2011. Associating the Visual Representation of User Interfaces With Their Internal Structures and Metadata. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, New York, USA.

[18] Web Accessibility Checker. 2011. IDI Web Accessibility Checker: Web Accessibility Checker. *Online. Available: http://achecker.ca/checker/index. php. [Last accessed: 28 10 2014]* (2011).

[19] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From Lost to Found: Discover Missing UI Design Semantics through Recovering Missing Tags. *Proceedings of the ACM on Human-Computer Interaction* CSCW (2020).

[20] 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 (Nov. 2019).

[21] 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 (ICSE '18)*. ACM, New York, NY, USA, 665–676.

[22] 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 Trans. Softw. Eng. Methodol.* 29, 3, Article 19 (June 2020), 31 pages. https://doi.org/10.1145/3391613

[23] 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. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020).

[24] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, and Liming Zhu. 2020. Object Detection for Graphical User Interface: Old Fashioned or Deep Learning or a Combination? *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020).

[25] 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).

[26] Sen Chen, Lingling Fan, Ting Su, Lei Ma, Yang Liu, and Lihua Xu. 2019. Automated cross-platform GUI code generation for mobile apps. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. IEEE, 13–16.

[27] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, 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*.

[28] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 767–776. https://doi.org/10.1145/2984511.2984581

[29] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. 2012. "Yours is Better!" Participant Response Bias in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1321–1330.

[30] Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1525–1534. https://doi.org/10.1145/1753326.1753554

[31] Morgan Dixon, James Fogarty, and Jacob Wobbrock. 2012. A General-Purpose Target-Aware Pointing Enhancement Using Pixel-Level Analysis of Graphical Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 3167–3176. https://doi.org/10.1145/2207676.2208734

[32] James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA.

[33] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. 2010. The Pascal Visual Object Classes (voc) Challenge.

International Journal of Computer Vision 88, 2 (2010), 303–338.

[34] MetalPop Games. 2020. UI Accessibility Plugin (UAP): GUI Tools: Unity Asset Store. https://assetstore.unity.com/packages/tools/gui/ui-accessibility-plugin-uap-87935

[35] Cole Gleason, Amy Pavel, Emma McCamey, Christina Low, Patrick Carrington, Kris M Kitani, and Jeffrey P Bigham. 2020. Twitter A11y: A Browser Extension to Make Twitter Images Accessible. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.

[36] Google. 2020. Android Accessibility Help: Switch Access. https://support.google.com/accessibility/android/topic/6151780.

[37] Google. 2020. Build more accessible apps. https://developer.android.com/guide/topics/ui/accessibility/.

[38] Google. 2020. Get started on Android with TalkBack - Android Accessibility Help. https://support.google.com/accessibility/android/answer/6283677?hl=en

[39] Darren Guinness, Edward Cutrell, and Meredith Ringel Morris. 2018. Caption crawler: Enabling reusable alternative text descriptions using reverse image search. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–11.

[40] Danna Gurari, Yinan Zhao, Meng Zhang, and Nilavra Bhattacharya. 2020. Captioning Images Taken by People Who Are Blind. (2020). arXiv:2002.08565

[41] Jaekyu Ha, Robert M Haralick, and Ihsin T Phillips. 1995. Recursive XY Cut Using Bounding Boxes of Connected Components. In *Proceedings of 3rd International Conference on Document Analysis and Recognition (ICDAR '95, Vol. 2)*. IEEE, 952–955.

[42] Vicki L Hanson and John T Richards. 2013. Progress on website accessibility? *ACM Transactions on the Web (TWEB)* 7, 1 (2013), 1–30.

[43] K He, G Gkioxari, P Dollar, and R Girshick. 2017. Mask R-CNN. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 2980–2988.

[44] MD Zakir Hossain, Ferdous Sohel, Mohd Fairuz Shiratuddin, and Hamid Laga. 2019. A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

[45] Forrest Huang, John F Canny, and Jeffrey Nichols. 2019. Swire: Sketch-based user interface retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–10.

[46] Andrej Karpathy and Li Fei-Fei. 2015. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3128–3137.

[47] 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.

[48] Ranjitha Kumar, Jerry O Talton, Salman Ahmad, and Scott R Klemmer. 2011. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2197–2206.

[49] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6038–6049. https://doi.org/10.1145/3025453.3025483

[50] 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* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 569–579. https://doi.org/10.1145/3242587.3242650

[51] 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.

[52] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03)*. Citeseer, 260–269.

[53] Kevin Patrick Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* (2018).

[54] George Nagy and Sharad C. Seth. 1984. Hierarchical Representation of Optically Scanned Documents. *Proceedings of the 7th International Conference on Pattern Recognition* (1984), 347–349. https://ci.nii.ac.jp/naid/10006753131/en/

[55] Alexander Neubeck and Luc Van Gool. 2006. Efficient non-maximum suppression. In *18th International Conference on Pattern Recognition (ICPR'06)*, Vol. 3. IEEE, 850–855.

[56] T. Nguyen, P. Vu, H. Pham, and T. Nguyen. 2018. Deep Learning UI Design Patterns of Mobile Apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. 65–68.

[57] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE, 248–259.

[58] Dan R. Olsen, Scott E. Hudson, Thom Verratti, Jeremy M. Heiner, and Matt Phelps. 1999. Implementing Interface Attachments based on Surface Representations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, New York, USA.

[59] Elaine Pearson, Chrstopher Bailey, and Steve Green. 2011. A tool to support the web accessibility evaluation process for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 28–32.

[60] Pixelogik. 2020. ColorCube. https://github.com/pixelogik/ColorCube

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

[62] Christopher Power, André Freire, Helen Petrie, and David Swallow. 2012. Guidelines are only half of the story: accessibility problems encountered by blind users on the web. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 433–442.

[63] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[64] 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.

[65] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2017. Epidemiology as a Framework for Large-Scale Mobile Application Accessibility Assessment. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility* (Baltimore, Maryland, USA) *(ASSETS '17)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/3132525.3132547

[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* (Galway, Ireland) *(ASSETS '18)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/3234695.3236364

[67] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2020. An Epidemiology-Inspired Large-Scale Analysis of Android App Accessibility. *ACM Trans. Access. Comput.* 13, 1, Article 4 (April 2020), 36 pages. https://doi.org/10.1145/3348797

[68] Santiago. 2018. Confusion Matrix in Object Detection with TensorFlow. https://towardsdatascience.com/confusion-matrix-in-object-detection-with-tensorflow-b9640a927285

[69] Richard S. Schwerdtfeger. 1991. Making the GUI Talk. ftp://service.boulder.ibm.com/sns/sr-os2/sr2doc/guitalk.txt.

[70] Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Amy J. Ko. 2018. Rewire: Interface Design Assistance from Examples. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 504:1–504:12.

[71] Amanda Swearngin and Yang Li. 2019. Modeling Mobile Interface Tappability Using Crowdsourcing and Deep Learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3290605.3300305

[72] Hironobu Takagi, Shinya Kawanaka, Masatomo Kobayashi, Takashi Itoh, and Chieko Asakawa. 2008. Social accessibility: achieving accessibility through collaborative metadata authoring. In *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*.

[73] Alibaba Tech. 2019. UI2code: How to Fine-tune Background and Foreground Analysis. https://medium.com/hackernoon/ui2code-how-to-fine-tune-background-and-foreground-analysis-fb269edcd12c

[74] Shari Trewin, Diogo Marques, and Tiago Guerreiro. 2015. Usage of Subjective Scales in Accessibility Research *(ASSETS '15)*. Association for Computing Machinery, New York, NY, USA, 59–67. https://doi.org/10.1145/2700648.2809867

[75] 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.

[76] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, and Jian Lu. 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* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2421–2436. https://doi.org/10.1145/3319535.3363193

[77] 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.

[78] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology* (Victoria, BC, Canada)

(UIST '09). Association for Computing Machinery, New York, NY, USA, 183–192. https://doi.org/10.1145/1622176.1622213

[79] Luke S. Zettlemoyer and Robert St. Amant. 1999. A Visual Medium for Programmatic Control of Interactive Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, New York, USA, 199–206.

[80] 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.

[81] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 609–621.