MotorEase: Automated Detection of Motor Impairment Accessibility Issues in Mobile App UIs

Arun Krishnavajjala
akrishn@gmu.edu
George Mason University
Farifax, VA, USA

SM Hasan Mansur
smansur4@gmu.edu
George Mason University
Fairfax, VA, USA

Justin Jose
justinjp2028@gmail.com
South Lakes High School
Reston, VA, USA

Kevin Moran
kpmoran@ucf.edu
University of Central FL
Orlando, FL, USA

ABSTRACT

Recent research has begun to examine the potential of automatically finding and fixing accessibility issues that manifest in software. However, while recent work makes important progress, it has generally been skewed toward identifying issues that affect users with certain disabilities, such as those with visual or hearing impairments. However there are other groups of users with different types of disabilities that also need software tooling support to improve their experience. As such, this paper aims to automatically identify accessibility issues that affect users with *motor-impairments*.

To move toward this goal, this paper introduces a novel approach, called MotorEase, capable of identifying accessibility issues in mobile app UIs that impact *motor-impaired users*. Motor-impaired users often have limited ability to interact with touch-based devices, and instead may make use of a switch or other assistive mechanism — hence UIs must be designed to support both limited touch gestures and the use of assistive devices. MotorEase adapts computer vision and text processing techniques to enable a semantic understanding of app UI screens, enabling the detection of violations related to four popular, previously unexplored UI design guidelines that support motor-impaired users, including: (i) visual touch target size, (ii) expanding sections, (iii) persisting elements, and (iv) adjacent icon visual distance. We evaluate MotorEase on a newly derived benchmark, called MotorCheck, that contains 555 manually annotated examples of violations to the above accessibility guidelines, across 1599 screens collected from 70 applications via a mobile app testing tool. Our experiments illustrate that MotorEase is able to identify violations with an average accuracy of ≈90%, and a false positive rate of less than 9%, outperforming baseline techniques.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Software design engineering**.

KEYWORDS

accessibility, mobile apps, screen understanding

ACM Reference Format:
Arun Krishnavajjala, SM Hasan Mansur, Justin Jose, and Kevin Moran. 2024. **MotorEase**: Automated Detection of Motor Impairment Accessibility Issues in Mobile App UIs. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639167

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639167

1 INTRODUCTION

The everyday lives of end-users depend on the use of software applications that support critical tasks such as banking, reading news, and communicating with others. Due to the central role of software in modern society, developers have an obligation to ensure that people of *all* abilities and backgrounds are able to use applications to carry out daily tasks. However, this ideal is still very much a goal that engineers must collectively work toward, as past work has illustrated the prevalence of accessibility issues in mobile app software ecosystems [23, 33, 88]. Furthermore, the need for accessible software now transcends a moral pursuit, as government agencies worldwide have begun to advocate for more accessible software by introducing legislation which, "prohibits discrimination on the basis of disability in the activities of public accommodations," [11]. Beyond providing equitable access to software for users with a variety of backgrounds, accessibility features often improve user experience more broadly, as many accessibility guidelines are designed following the general principals of universal design [8], in that the adherence to such guidelines is more likely to lead to an improved user experience for *all* users [83].

Current research at the intersection of developer tools and software accessibility has generally been disproportionally focused on users with certain disabilities, such as visual impairments, i.e., low vision (LV), and hearing impairments, i.e., deaf and hard of hearing (DHH) [33, 40, 73, 88, 91]. The visual nature of software user interfaces (UIs), and large populations of users with visual impairments have made this a natural and important focus area. This focus, however, must expand to study and create tools that aid developers in considering and implementing accessible features that support users with a wider range of disabilities. One understudied demographic, and the focus of this paper, is that of *motor-impaired users*. The current landscape of research on developer tools that aim to support software accessibility for motor-impaired users is somewhat limited, due in part to the difficulty in supporting a wide spectrum of motor-impairment conditions (i.e., ranging from hand tremors, to more limited motor abilities that necessitate the use of assistive devices such as switch controls) and need to consider external hardware [83] . Generally, developers currently lack tools for identifying, understanding, and implementing accessible features for motor-impaired users [23].

The central challenge of building developer tools that support accessibility for motor impaired users is one of *semantic screen understanding*. That is, in order to determine whether a given UI screen follows motor-impairment accessibility guidelines, the *functional*

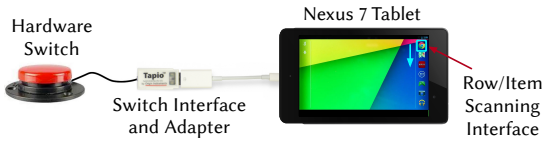

Figure 1: Switch Interface

and *visual* properties of UI screens and individual components must be automatically inferred from a given app. For example, motor-impaired users that make use of hardware devices, such as switches, rely on assistive services (e.g., Android `Switch Access` [3]) that iteratively scan through and highlight individual UI components, as illustrated in Figure 1. This allows a motor-impaired user to easily select the icon or component with which they wish to interact. However, this process can be slower than traditional gesture-based control [55], and switch users often rely on the *consistency* of certain UI element patterns, such as menus, in order to quickly perform actions. As such, a popular motor-impairment accessibility design guideline for mobile apps [1, 13] states that persistent icons that appear across multiple screens, such as tab bars or menus, should retain a consistent ordering. This allows a user to anticipate which UI elements will be highlighted by the assistive service. However, in order to detect inconsistent icon orderings, an automated tool must be able to accurately identify functional groups of UI components, such as tab bars, and the ordering of icons within them.

To help advance the current state of developer tools to better support motor impaired users, and overcome the challenges related to automated screen understanding, this paper introduces a novel approach, called MotorEase, which aims to automate the detection of **Motor** impairm**E**nt **A**cce**S**sibility issu**E**s in mobile apps. MotorEase is a novel approach that leverages automated dynamic analysis, computer-vision, and text-processing techniques to detect violations of motor-impairment accessibility guidelines in a given Android application. The approach is comprised of four detectors each targeted toward a popular UI design guideline meant to support motor impaired users. MotorEase's novelty lies in both its technical underpinnings and its ability. MotorEase combines multiple neural models for screen understanding, allowing it to recognize screen semantics prior techniques cannot. This allows MotorEase to identify violations of motor-impairment accessibility guidelines. MotorEase is designed to seamlessly integrate into existing software testing workflows, and operates in a fully automatic manner by analyzing common artifacts produced by existing Android testing tools. MotorEase then passes this data to a series of four detectors, each of which identifies guideline violations. The main contributions of this paper are as follows:

(1) MotorEase – a novel, automated motor-impairment accessibility guideline violation detection tool which is able to accurately detect issues in applications when used in conjunction with an automated inout generation tool;
(2) The MotorCheck benchmark, consisting of 555 screens that violate one of four popular motor-impairment accessibility guidelines, and 1044 that follow such guidelines, collected from popular Google Play applications.
(3) The results of an empirical evaluation that measures the effectiveness of MotorEase in detecting motor-impairment accessibility issues within the MotorCheck benchmark.
(4) A Replication Package [5–7] that contains MotorEase's code, the MotorCheck benchmark, and our experimental infrastructure to foster replicability and further promote research on motor-impairment accessibility issues.

2 BACKGROUND AND MOTIVATION

In order to understand how the MotorEase approach functions, it is important to understand how motor-impaired users use their devices. Motor-impaired users use devices in two main ways: Switch input and touch input [92]. As described earlier, switch based input uses an external hardware input device. Switches are common in users with limited to no motor control but without impaired cognitive functions [92]. Most current smartphone and tablet devices offer accessibility services that support scanning a curser across the screen to highlight UI elements and icons (*e.g.*, Fig. 1) [1, 13]. The users can then click the switch to select the UI element that is highlighted, forming a semi-automated form of user input. This input technique can be far slower than traditional touch-based interactions, as it requires waiting for appropriate UI elements to be highlighted by the system. Thus, switch users rely on consistent and accessible app designs to be able to anticipate the future UI elements that will be focused by the scanning service. One example that highlights how tedious this input can be is related to typing speed, where switch users average 3 words per minute (wpm), and touch-based users average closer to 40 wpm [55].

For users with some, but limited motor-control, touch input is typically used with constraints related to physical limitations such as tremors or gesture speed limitations [63]. Some users with lesser motor-impairments may also turn to using voice commands to navigate applications due to difficulties/slowness related to switch operation [92]. Prior work in the HCI research community has aimed to improve gesture recognition for users with limited motor control [75]. Google and Apple also offer sensitivity and touch settings that aim to allow users with limited motor control to customize touch controls for easier interaction [1, 13, 75]. Google and Apple [1, 13] have also developed UI and human interaction guidelines which aim to support motor-impaired users, but as past work has illustrated, developers may often be unaware of such guidelines or ignore them due to other development constraints [80].

2.1 Accessibility Guideline Literature Review

In order to fully capture the current landscape of accessibility guidelines that may impact various populations of users, and to aid in selecting the most impactful guidelines that aim to assist motor impaired users we conducted a systematic literature reviews on research at the intersection of software engineering, human-computer interaction, and accessibility. To conduct this review, we followed the methodology set forth by Kitchenham *et al.* [44]. We defined a single research question that asked *"What accessibility guidelines have been identified and discussed in prior research?"*. We used the relatively simple search string of "accessibility" to search DBLP, the ACM Digital Library, and IEEE Xplore, for work at the intersection of accessibility and software engineering for the date range of January 2010 - December 2022. The purpose of using such a simple search string was to "cast a wide net" and ensure that we did not miss important work. We defined inclusion criteria as follows: (i) must have been published in our studied date range, (ii) must

Table 1: Accessibility guidelines extracted from our systematic literature review of accessibility guidelines – includes recent research and Google's [13] and Apple's [1] accessibility guidelines. (LV = low vision users, DHH = deaf and hard of hearing users)

Accessibility Guideline	Primary Affected User Demographic	Guideline Source	Previous Implementation	Implemented by MotorEase
Visual Touch Target Size	Motor, LV	[45, 72]		✓
Touch Target Size	Motor, LV	[1, 13, 14, 16, 20, 23, 30, 43, 45, 70]	✓	
Persistent Element Location	Motor	[1, 13, 14, 16]		✓
Clickable Span	Motor, LV	[23]	✓	
Duplicate Clickable Bounds	Motor	[23]	✓	
Editable Item Descriptions	LV	[23, 38]	✓	
Expanding Section Closure	Motor, LV	[1, 13, 14, 16]		✓
Non-Native Elements	Motor, LV	[13, 30]	✓	
Motion Activation	Motor, LV	[1, 13, 14]		
Labeled Elements	Visual, DHH	[23, 38, 39, 46]	✓	
Screen Captioning	LV, DHH	[1, 10, 12–14, 16, 43, 46, 74, 79]	✓	
Keyboard Navigation	Motor, LV	[10, 12, 35, 39, 46]	✓	
Traversal Order	Motor	[12, 23, 39]	✓	
Adjacent Visual Icon Distance	Motor	[1, 13, 16, 20, 70, 90]		✓
Proper Information Organization	Motor, LV	[30]	✓	
Facial Recognition	Motor	[24, 30]	✓	
Single Tap Navigation	Motor	[1, 13, 14, 16, 39, 61]		
Poor form design/instructions	Motor, LV	[10, 12]		

have been published at one of 16 conference venues (ICSE, FSE, ASE, ICSME, MSR, ICPC, ISSTA, ICST, SANER, UIST, CHI, SPLASH, OOPSLA, PLDI, CSCW, ASSETS) or 5 journal venues (TSE, TOSEM, EMSE, JSS, ASE) that cross cut software engineering, HCI, and accessibility, (iii) the paper must describe a study or developer tool directly related to an accessibility issue that impacts end-users. The scope of our search was limited to these venues and digital libraries as they provide the highest quality of research in all matters including accessibility. Our search results returned 2948 papers from our selected conferences within our given date range. Then, two authors manually checked each paper for adherence to the final inclusion criteria, resulting in 20 papers that intersect our desired research areas *and* discuss developer guidelines for addressing accessibility issues. In addition to these 20 identified primary studies, we also examined Apple's and Google's design guidelines related to accessibility [1, 13], as several of our primary studies referenced these sources.

After the search process concluded, one author extracted all accessibility guidelines discussed in the papers and platform documentation, and after the process, three authors met to discuss and verify that all guidelines were properly extracted in a joint meeting. This process resulted in the derivation of 18 *accessibility design guidelines*, which are illustrated in Table 1. In this table, we provide (i) a short description of the guideline (with full definitions and examples available in our online appendix), (ii) the primary affected user groups, (iii) the sources that described the guideline, (iv) whether or not automated support for guideline has been implemented in past commercial or research developer tools, and (v) the guidelines targeted by MotorEase. **It should be noted that none of the guidelines that MotorEase targets have been explicitly targeted by prior tools.** While touch-target size has been explored in prior work [13, 20, 23, 43], *visual* touch-target size, which is of critical importance for motor-impaired users [45], has not been explored. While there is some recent work on Keyboard Accessibility failures in web applications that could affect Motor-impaired users [34], this work does not explicitly target any of the guidelines targeted by MotorEase. The Groundhog [81] tool is also inadvertently able to detect *some* expanding section closures. The lack of exploration of these guidelines is largely due to the fact that implementing tools that detect when such guidelines are *not* followed requires new types of automated UI screen understanding.

The four accessibility guidelines targeted by MotorEase are (i) ***Visual Touch Target Size***, (ii) ***Persistent Element Location***, (iii) ***Expanding Section Closure***, and (iv) ***Visual Icon Distance***. One of these guidelines (*Visual Touch Target Size*) was identified from a prior accessibility study, whereas the others come directly from Apple and Google's accessibility deign guidelines for mobile apps [1, 13]. These four guidelines were chosen as they had not been implemented by past work and could easily integrate with automated input generation (AIG) tools for Android, which is an important practical component of MotorEase as we explain in Section 3. In the following subsections, we describe each motor-impairment accessibility guideline in detail, and the screen understanding challenges in detecting guideline violations. More detailed descriptions of all guidelines can be found in our online appendicies [5–7].

2.2 Expanding Section Closure

Pop up menus and modal dialogs can provide meaningful information to the user, but closing them can be non-trivial for motor-impaired users who utilize a switch, as they may not contain explicit UI elements for closing the menu or dialog. As such, UI design guideline advocated for by both Apple and Google [1, 13] state that such closure UI elements should be present and easily interactive. Many expanding sections can be closed through a swiping gesture to dismiss the section or an external tap on the screen not within the bounds of the expanding section. Both of these options pose an accessibility issue for motor-impaired users due to the need for gestures and assumptive tapping on non-intuitive screen locations. A violation and adherence to this guideline is illustrated in Figure 2-①. Detecting UIs that violate this design guideline can be difficult as it requires automatically identifying (i) whether a pop-up menu or modal dialog is present within a given screen, and (ii) whether or not a UI element supports closing the pop-up.

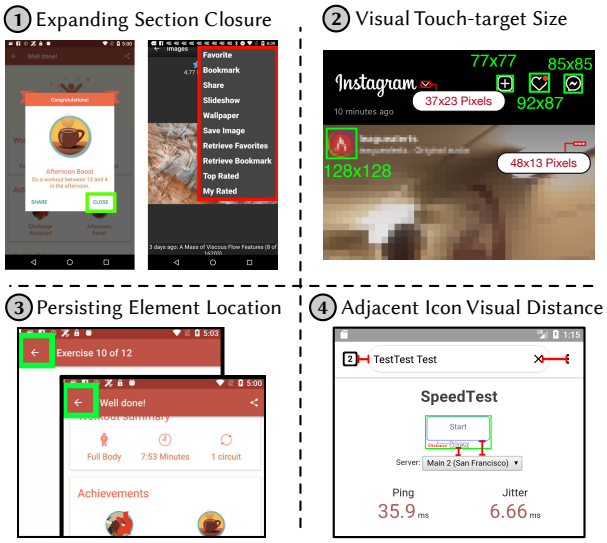

Figure 2: Illustration of four studied accessibility guidelines

2.3 Visual Touch-Target Size

Motor-impaired users who experience tremors in their hands can experience difficulty tapping precisely on icons. This makes it difficult for them to interact with elements as intended. Apple and Google suggest minimum UI element sizes of 44x44 pixels and 48x48 pixels respectively, such that users with minor motor impairments can more easily tap icons [1, 13]. Typically, the "size" of a UI element is defined by the *touchable area* of that element, and not the *visual area* occupied by the pixels of a given element. However, as stated above, it is important to provide sizable *visual* touch targets to users with more limited motor control, such that they can better hone their more limited movements to tap desired UI elements. Most past work that aims to identify icons or UI elements that do not meet a given minimum threshold read UI metadata to examine the "touchable area" only, even if the visual size of the icon does not fill the entire area. Thus, this can create a disconnect between the *visual* touch target size, and the *touchable area*. An example of this is shown in Figure 2-②. The ✉ icon next to the Instagram logo has a touch target size of larger than 44x44, however, the visual size of this icon is quite small, a making it difficult to tap.

2.4 Persistent Element Location

Applications link various screens together to aid users in completing complex tasks, however, certain UI elements need to exhibit *consistent* placement to assist switch users with anticipating UI element scanning. This guideline specifies that icons that appear across multiple screens should appear in the same general area of the screen [1, 13]. This means the locations of elements such as back buttons or search icons that appear across multiple different screens should remain consistent. An example of a back button with a consistent location is illustrated in Figure 2-③. Violations of this guideline can be difficult to detect as it requires automatic identification of corresponding UI elements across screens which may exhibit visual variability (e.g., displayed on different backgrounds).

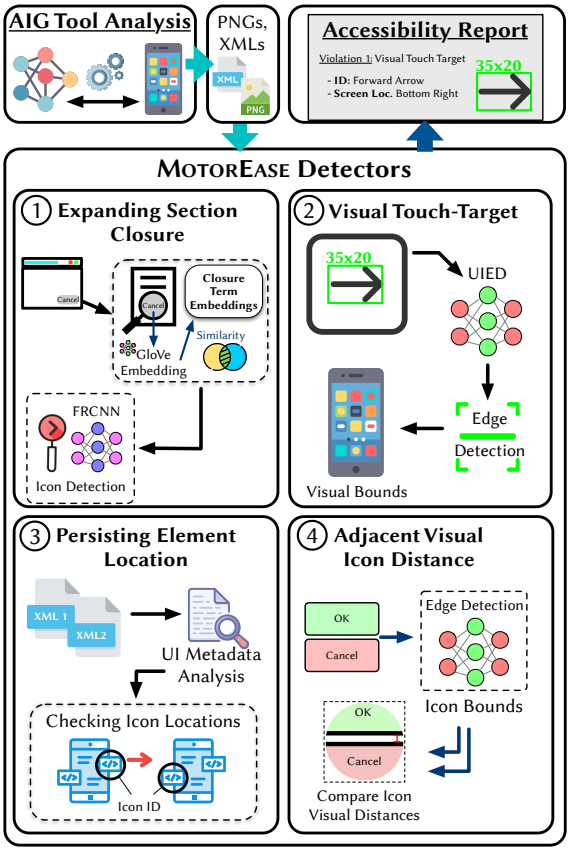

Figure 3: Overview of MotorEase's Workflow

2.5 Adjacent Visual Icon Distance

The design and placement of interactive icons that signal functional affordances is critical to ensuring a positive user experience, particularly for individuals with motor impairments. For users who may struggle with fine motor movement, it can be difficult to tap a single location on the screen without accidental triggers of other areas [45]. As such, the *Adjacent Visual Icon Distance* guideline states that adjacent "clickable" UI elements should be positioned at least eight pixels apart from one another. This can be challenging as it again requires the automated inference of the visual area occupied by different UI elements. An example of this is shown in Figure 2-④, wherein the "Start" and "Privacy" elements are located too close to one another and may result in accidental, unintentional triggering of the components by a motor-impaired user.

3 THE MOTOREASE APPROACH

MotorEase is an automated approach that aims to detect motor-impairment accessibility guideline violations by analyzing UI metadata and screenshots collected via automated input generation (AIG) tools (i.e., automated app crawlers, UI testing tools). MotorEase operates in three stages, and implements four guideline violation detectors, as depicted in Figure 3. First, an AIG tool is run on a target application to produce a set of screenshot and `uiautomator` XML files (*i.e.*, UI metadata) before and after each AIG tool action. We tailor our approach to utilize UI metadata generated using the `uiautomator` framework, which captures UI layout information

Table 2: Mapping of Sample Lexical Patterns to detect Closure

Initial Closure Words
"close", "cancel", "dismiss", "done", "ok", "finish", "return"
GLoVE Embedding Words
"deny", "allow", "exit","end","terminate","quit","back","stop","ignore", "proceed","save","apply","submit","confirm","abort","decline","reject","ignore"

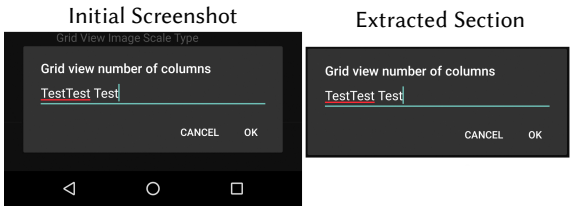

Figure 4: Extracted Expanding Sections

in a structured XML format, as this is most prevalent utility used by recent Android AIG tools [41, 48, 50–52, 57, 65, 66, 86, 93]. It should be noted that MotorEase does not require any pre-existing test cases, but instead can be used in conjunction with any of the AIG tools listed above. Second, MotorEase utilizes a series of four *violation detectors* to analyze the screenshots and UI metadata to determine if the target application failed to follow motor-impairment guidelines. Finally, MotorEase collects the information from the detectors and compiles an *accessibility report* that informs developers of accessibility guideline violations.

3.1 Detectors

The core components of MotorEase are its four accessibility guideline violation detectors. Detectors ①, ② and ④ operate upon *single* `uiautomator` XML files, screenshots, in the form of PNGs, or both. Detector ③ takes as input a series of *multiple* XML and screenshot pairs. In the remainder of this section, we describe the technical underpinnings of each of MotorEase's detectors.

3.1.1 ***Expanding Section Closure Detector.*** The expanding sections detector aims to identify pop up messages or slide-in views that lack a visible means to close the section. Objects or text that imply closing the section is what MotorEase aims to detect, if it cannot detect these, then a given screen with a dialog box or section is considered to be in violation of the guideline. This detector begins by determining whether the screen has an expanding section and then extracting it from the screenshot. This is done by identifying the largest element on the screen. MotorEase extracts the largest *android.widget.FrameLayout* and the largest *android.widget.ListView* on the screen whose size is not the entire screen as the pop up screen or the slide-in list menu. An example of this is shown in Figure 4.

Once MotorEase has extracted each of the expanding sections, it then aims to determine whether the pop-up or section provides a clear means of closing it. In order to offer a robust solution, MotorEase accomplishes this via two main procedures: (i) Semantic Text Matching and (ii) Icon Detection. This is due to the fact that closure controls can have either textual (*i.e.*, the word "exit") or visual (*i.e.*, an × icon) signifiers that indicate functionality.

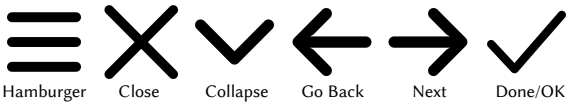

Figure 5: Base Closure Icons

Semantic Text Matching: MotorEase's semantic matching technique defines a certain set of keywords that are likely to signify an element that can close an expanding section or pop-up. These keywords comprise common lexical patterns derived through two authors of the paper examining expanding sections that appear in 1500 randomly sampled screenshots from the RICO dataset [37]. The RICO dataset, comprising 9,000+ Android apps and 66,000+ screenshots, serves as a popular resource for mobile app research given its diverse set of screens and apps. We randomly sampled 1500 screens as it represents a statistically significant sample size of the 66k screenshots present in the RICO dataset (95% confidence level and 2.5% margin of error). These words were terms that implied a closure or completion action, i.e. "close", "dismiss", "cancel", "ok". To ensure that the selected words for semantic matching were relevant, we employed a manual process in which two authors reviewed the randomly sampled dataset for expanding elements and words that implied closure. The resulting set of words was agreed upon by the authors as suitable for the task. To further expand this set of words, we further utilized GloVe embeddings [76] to compare the original selected words to the entire dataset. Glove embeddings capture semantic relationships between words by considering global word co-occurrence patterns, resulting in dense vector representations that preserve meaningful similarities between words [76]. We extracted additional words that exhibited a cosine similarity of at least 0.95 to the GloVe embeddings of the original selected words. This approach enabled us to carefully curate a comprehensive set of words for semantic matching that accurately represented closure. We provide the complete list of 25 closure words in Table 2 – as our experiments illustrate, we found these words to generalize well to our experimental benchmark.

To extract the text from each image, MotorEase leverages a combination of Google's Optical Character Recognition (OCR) text extraction [4], which is based on the EAST OCR technique [94] and the text present in the `uiautomator` XML file. We use both methods of text extraction as text displayed on the screen via images is often not captured in the `uiautomator` XML files. This method provides a binary classification for the presence of text that indicates a means to close the pop-up. If there are no matching extracted terms, MotorEase then proceeds with icon detection.

Icon Detection: Icon detection is used to identify specific closing icons, i.e. ×, hamburger icon, checkmark, etc. Two authors examined the same set of 1500 screens from the RICO dataset discussed above and compiled a set of base icon types to train an image detection model in order to detect these icons. The chosen base icons are shown in Figure 5. To accurately perform this detection, we trained a neural object detection model on a diverse set of examples of these identified icons, through a process we describe in detail below.

Training a neural object detection model typically requires a large-scale dataset with annotated examples of the target icons. To derive such a dataset, we *automatically* constructed a realistic, synthetic dataset of icons that represent menu/pop-up closing. To

do this, we extracted icon images with transparent backgrounds from the Fontawesome [1] and Flaticon [2] image repositories until five icons for each icon type were identified that varied in color and style. Note that these icons do not directly appear in our evaluation dataset. We then superimposed these icons into random locations on screenshots derived from the RICO dataset [37]. During the analysis of the 1500 RICO screenshots two authors determined that a majority of icons on the screen were mathematically smaller than 10% of the total screen size and larger than 2% of the screen size. The screen size in question are the 1920x1080 pixel dimensions of the screen, limiting the maximum icon size to 192px and minimum icon size to 38px. Therefore, when an icon was superimposed on a screen, we varied their size between 2% and 10% of the screen area so that they remained relatively similar to the screens in the dataset. In this manner, we generated a dataset of 7,291 images (separate from the 1500 images sampled earlier) all with labeled and fully-localized icons on them (one per screen, spread evenly across the variations of our six icon types) and divided this into training and testing sets following an 80/20 split, 5,832 images for training, 1,458 for testing. We then used this dataset to train a Faster-RCNN object detection technique [78] using the the torch-vision API [15]. We generated ≈ 7k images as past work that uses a similar approach for icon detection was able to train an accurate model with this scale of data [27, 28, 42]. Our trained model was able to achieve over 95% accuracy on the test portion of our dataset.

MotorEase passes the cropped out expanding sections to the model in order to detect the icons. This detector works by first checking for semantic text matches and if there is no match, it then checks for icons. MotorEase performs text pattern matching first because of the potential for X icons on the expanding section *not* related to closing the icon. Had we done the icon detection first, the X icons to delete text in the text-fields would have been detected by the object detector. This means that this screen would have been classified as closable even though the X icons do not imply closing of the section. Therefore we use both the text and images on the expanding sections to try and classify if it can be closed. If neither technique can pick up on a pattern or icon, MotorEase reports the section as a violation to the expanding section guidelines, capturing the the screenshot name, FrameLayout/ListView name, and violation and make it known to the developer.

3.1.2 ***Visual Touch-Target Detector.*** This detector uses both the screenshot and its corresponding XML file. It starts by processing the XML file and extracting the bounds for all elements which are clickable. XML has various properties in its metadata to describe an element, and if an element has a *"True"* in the *"clickable"* field, we determined that it is meant to be clicked on or interacted with. This detector aims to identify elements that have a visible area that is smaller than their tappable/clickable bounding box. Hence with this detector, MotorEase aims identify elements whose *visual* size are under 48x48 pixels [9], even when the reported touchable area (as indicated by `uiautomator`) may be larger than 48x48. The bounding boxes in `uiautomator XML` files can show a bounding box whose size is larger than the actual size of the icon. An example of this is shown in Figure 6. The true bounds of an icon corresponds

[1] https://fontawesome.com
[2] https://www.flaticon.com

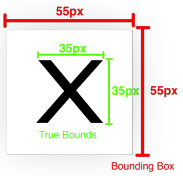

Figure 6: Example of Bounding Box vs. True Bounds

to the visible area occupied by the icon. This can make these bounding boxes appear to be guideline abiding since they are generally larger than the true, visible bounds of the icon they hold. In order to identify these cases, MotorEase adds 15 pixels to the width and height of the bounding boxes of each clickable item to extract the entire icon, before cropping the enlarged icon from the image.

After MotorEase extracts the element, it is used as input to an edge detection algorithm, implemented in the UIED tool [68], which is an approach that combines both neural object detection and unsupervised edge detection to effectively segment mobile app UI screens. This edge detection procedure is able to derive the *visual* bounds of a given UI element, and would identify the 35x35px edges for the "X" as shown in Figure 6. This procedure is applied to clickable icons extracted on each screen. Once the true edges are identified, MotorEase is then able to then able to compare these bounds to the reported element size in the `uiautomator XML` file. If it is determined that the true element width or height is less than 48 pixels, MotorEase labels that individual icon as a violation.

3.1.3 ***Persisting Elements Detector.*** The persisting elements detector aims to identify an icon on a screen whose functionality remains the same, but location changes across multiple screens. An example of this is the navigation bar at the bottom of most applications. We expect that if the navigation bar is at the bottom of the screen on one screen, if a second screen has a navigation bar it should also be at the bottom of the new screen. This detector requires the use of both screenshots and `uiautoamtor XML` files from multiple screens, as MotorEase directly analyzes properties of UI elements in the `uiautoamtor XML` files and compares the visual UI element similarity across multiple files.

This detector parses all of the `XML` files for a given application and records all of the UI element IDs. Then, it collects the location(s) across all `XML` files for each individual element ID. If there was more than 1 instance of a given ID across multiple screens, MotorEase checks to determine whether the location bounds were the same as well as checking if the elements within those bounds are visually similar (95% similar according to pixel-based mean squared error). If they are not, it deems it a violation of the persisting element guideline. This ensures that MotorEase is checking every element in the application while checking to see if the icons with similar IDs have similar visual properties. If there is an element that appears across the `XML` files more than once, MotorEase examines them to see if the location is the same for the element. It relays this information back to the developer by providing the ID of the violating element in the generated Accessibility report.

3.1.4 ***Visual Icon Distance Detector.*** The icon distance detector is designed to analyze icons on a screen and determine whether the visual distance between any two icons is less than 8 pixels,

which combines findings from Kong *et al.* [45] regarding visual icon size/spacing and recommendations from Google's accessibility Guidelines [13]. This detector analyzes a XML and screenshot pair, and extracts all of the *clickable* elements on the screen using the `uiautomator` metadata. Once the clickable elements are identified, MotorEase crops out the icons and sends them as input to UIED [68] to find the true visual bounds of the icon via edge detection. The true bounds are then used to subtract the padding from the initial touch target to the visual touch target. These modified bounding box values are then stored, and the detector iterates through all of the other icons on the screen to calculate the distance between the target UI elements and all other elements. The distance between the bounding boxes is calculated by determining the horizontal, vertical, or diagonal distance in pixels between two bounding boxes on a screen. Once the distance between the icons is computed, MotorEase checks whether there is a distance is less than 8 pixels. If a violation is detected, MotorEase includes a description of the UI elements in the generated accessibility report.

3.2 Accessibility Report Generation

The accessibility report is generated using the target app screenshots that contain violations and a generated markdown file, with filenames to specific images of violations for each detector. We developed textual templates for each type of violation that are used by the report generation engine to automatically describe the accessibility violations by filling in the templates with information the MotorEase analysis. This accessibility report aims to provide a comprehensive account of accessibility issues within a given app so that developers are able to take this information to make any changes they may need in order to make their apps more accessible.

4 DESIGN OF EMPIRICAL EVALUATION

In this section, we describe the procedure we used to evaluate MotorEase. To achieve our study goals, we formulated the following five research questions:

- **RQ$_1$** *How accurate is the Expanding Section detector?*
- **RQ$_2$** *How accurate is the Visual Touch Target detector?*
- **RQ$_3$** *How accurate is the Persisting Element detector?*
- **RQ$_4$** *How accurate is the UI Element Distance detector?*
- **RQ$_5$** *Does* MotorEase *identify a limited number of false positive and negative violations?*

4.1 RQ$_1$ - RQ$_4$: Violation Detection Capability

MotorEase is one of the first tools to support the detection of violations of accessibility design guidelines targeting motor-impaired users, and accomplishes this via understanding the visual and textual modalities Android UI screens. Given that prior techniques are not able to explicitly detect the design violations that MotorEase targets, by using the visual comprehension of the screen, we both derived an entirely novel benchmark and designed an evaluation methodology that tests each of its four detectors individually to determine their accuracy, precision, and recall. In addition, we also examine the false positive and false negative rates to better understand the practical utility of MotorEase. The evaluation metrics used in this study provide insights into the ability of each detector to identify true positives and true negatives. Accuracy gives us an overall ability to deduce each detectors ability to detect true positive (TP) and true negative (TN) values accurately. Note that we balance the positive and negative samples in the MotorCheck benchmark to allow for an informative accuracy measurement.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

In the context of our study, a True Positive (TP) is defined as the detection of an existing design guideline violation as defined in our ground-truth dataset. A false positive (FP) is defined as the detection of a violation when one does not exist. A False Negative (FN) occurs when our approach does not report a violation, but one exsists in the ground truth. Finally, a True Negative (TN) occurs when the approach does not report a violation and one does not exist. In addition to accuracy, we also measure the precision and recall (as defined below) to provide a more complete picture of MotorEase's performance. The results of MotorEase were manually validated (*i.e.*, two authors compared MotorEase's output to the ground truth.)

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$

Given these two evaluation metrics, we can determine how accurately each detector works. MotorEase's overall effectiveness can be derived by averaging the the accuracy/precision/recall across all four detectors, providing a comprehensive understanding of MotorEase's ability to detect accessibility guideline violations.

Table 3: Expending Sections Detector Test Dataset

Total Files	Close: Icon	Close: Text	Cannot Close
483	27	214	242

Table 4: Visual Touch-target, Persisting Elements, and Visual Icon Distance Test Datasets

Detector	Total Files/Apps	Violations	Non-Violations
Touch-Target	400 files	176	224
Persisting Elements	49 apps	24	25
Icon Distance	400 files	42	358

4.2 RQ$_5$: MotorEase's Practical Utility

In order to investigate MotorEase's practical utility, we also report both the false positive and false negative rate, as these reflect the need for a developer to sift through incorrect violation reports, or lost quality in terms of miss unreported violations. This helps to provide a more holistic picture of MotorEase's performance.

$$FalsePositiveRate = \frac{FP}{FP+TN} \qquad FalseNegativeRate = \frac{TN}{FP+TN}$$

4.3 Derivation of the MotorCheck Benchmark

Given that no prior approach has targeted the motor-impairment design violations targeted by MotorEase, we develop a novel benchmark called MotorCheck which we discuss below. It should be noted that all of the accessibility violations of this benchmark are real, no synthetic violations were injected in it's construction – instead real violations were rigorously manually annotated.

To derive the initial set of screenshots and xml files for MotorCheck we applied the CrashScope [65] automated testing tool to 70 popular Android apps that are cross-listed on both FDroid and Google Play. To do this, we gathered a list of apps from F-Droid [17] and considered only those apps that were cross-listed

on Google Play [18] and had at least 1000 downloads – providing some confidence in the popularity of the chosen applications. We provide a full list of these applications with download statistics and links in our online appendicies [5–7]. During this process, we used one of CrashScope's exploration strategies to extract 2,864 screenshot/XML pairs. Note that the goal of our study in assessing MotorEase's capabilities is independent of the coverage provided by the underlying testing tool, although CrashScope has been illustrated to be competitive with other tools [66]. It should be noted that the screen coverage of MotorEase is dependent upon the Android AIG tool that the approach is paired with. Given recent advances in AIG tools [89], MotorEase can integrate with these new tools and take advantage of the improved coverage. Next, we describe the dataset derivation process for each detector. Note, given that the data labeling process is quite objective for violations of our identified guidelines, for each dataset, we had one author manually label each instance, and another author verified the results. During this process, no instances of conflicts were noted, again due to the largely objective nature of the labeling procedure. We provide an overview of the MotorCheck benchmark data in Tables 3 & 4.

4.3.1 Expanding Section Closure Detector Dataset. In order to detect expanding sections, this detector requires an input screenshot and its corresponding XML file. In order to evaluate the detector and remove any bias, one author labeled expanding sections without closure elements until all CrashScope files were exhausted, resulting in 241 screens. Of the 241 screens that had an expanding section, there were 121 screens were FrameLayouts and the remaining 120 were ListViews. Then, in order to balance the dataset, an additional 242 screenshots without violations were randomly selected to complete the dataset, for a total of 483 screens. The additional screens without violations consisted of expanding sections that could be closed. Table 3 provides more information on how the dataset was split between violations and non-violation samples. Labeling was done manually using LabelStudio [19]. For the screenshots without violations, icon types and closure word types were labeled. If neither the icon nor text clearly showed a means of closing the section, it was labeled as a violation.

4.3.2 Visual Touch-Target and UI Element Distance Detector Dataset. The touch-target detector requires screenshot and XML pairs to determine if the screens had a touch-target violation (*i.e.*, the XML) and visual bounds differed. In order to evaluate the detector and remove any biases, we randomly chose 400 screenshots and XML pairs generated by CrashScope stratified across our 70 applications. This sample size was used as it represents a statistically significant sample of the total number of extracted CrashScope screens (99% confidence interval). Table 4 provides the dataset splits between violations and non-violation samples. Labeling for these screenshots was performed manually. One author analyzed each of the screenshots and set a bounding box on each of the interactive elements on the screen using Label Studio [19]. If the size of the bounding box was less than 48 in width or height, it was labeled it as a violation, else it was labeled as a screen without violations. Additionally, the author checked the distance between all components on these screens and labeled any instances where UI elements were less than 8 pixels apart. This was done for all 400 images.

4.3.3 Persisting Elements Detector. The persisting elements detector requires multiple app XMLs in order to detect violations. The CrashScope dataset [65] contains 70 applications and their screenshots. We filtered out apps that only contained screenshots of similar screens, resulting in 49 applications, 24 of which had persisting elements that violated our guideline, and 25 of which adhered to our guideline. One author labeled each app as having a violation or not having a violation. During the labeling, this author also specified which specific screenshot exhibited the violation.

4.4 Comparison to Baseline Techniques

While the Accessibility issues that MotorEase targets have not been explicitly targeted by past tools, there are two tools which are capable of detecting a subset of the accessibility violations identified by MotorEase. These two baselines are Groundhog [81] and Google Accessibility Scanner [2]. We ran these two tools on the same MotorCheck benchmark used to evaluate MotorEase to keep the comparison fair and consistent, and we report the same metrics for both MotorEase and the baseline techniques. Upon careful analysis of these baselines, Google Accessibility Scanner and Groundhog are only capable of detecting Touch-target size violations and Expanding Sections violations, respectively.

The first baseline tool we compared MotorEase to in our updated evaluation is Google's Accessibility Scanner [2]. This tool operates directly upon the dynamic representation of the GUI as reported by `uiautomator`, and checks to whether the bounds of screen elements fall below the 48x48 dp threshold. To apply this tool, we launched the each app included in the MotorEase dataset on an emulator of the same screen dimension (1920x1080) and Android version as the screens in the MotorCheck benchmark and manually navigated to the screen in question, and triggered the Accessibility Scanner tool. One author then manually compared the output results from Accessibility Scanner to the ground-truth visual touch-target size violations defined in MotorCheck. We then reported the Precision/Recall/Accuracy and F1 Score Metrics.

The second baseline tool that we compared MotorEase to is the Groundhog Accessibility tool [81]. We worked directly with the authors of the Groundhog tool, who were quite helpful after some initial issues initializing the tool, in order to apply Groundhog to the MotorCheck benchmark. Groundhog functions by first sending touch-based actions to a given Android app screen running on an emulator or real device, and then attempts to exercise the same actions using one of Google's accessibility services, such as Talkback. Given the manner in which Groundhog works, the only Accessibility issue in MotorCheck that it was applicable to is the Expanding Section guideline violations. To apply Groundhog to detect Expanding Section violations in MotorCheck we launched a target on an Android emulator configured to the 1920x1080 screen size and Android version of the MotorCheck screens, and then ran Groundhog on the screen with an expanding section. Groundhog then navigated the screen both with and without assistive services to determine whether it could close the expanding section, if there was a discrepancy, this was reported by Groundhog. Then one author manually checked the output of Groundhog to determine if it was able to detect an expanding section violated the MotorCheck

Table 5: Overall Results for MotorEase Detectors & Baselines

Approach	Precision	Recall	Accuracy	F1-Score
MotorEase (Viz T.-Target)	1.0000	0.6648	0.8525	0.7986
G-Accessibility Scanner (T.-Target)	0.5556	0.5085	0.6025	0.5310
MotorEase (Exp. Sec)	0.9042	0.9205	0.9123	0.9129
Groundhog (Exp. Sec)	0.6849	0.8659	0.7207	0.7648
MotorEase (Pers. Elem)	0.8214	0.9583	0.8776	0.8846
MotorEase (Icon Dist)	0.7119	1.0000	0.9575	0.8317
MotorEase (All Detectors)	**0.8594**	**0.8859**	**0.8999**	**0.8570**

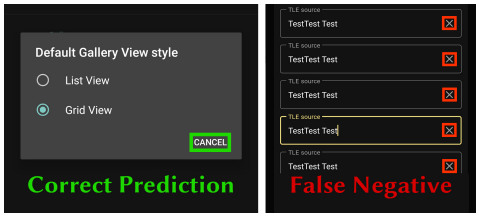

Figure 7: Expanding Section Closure Detection

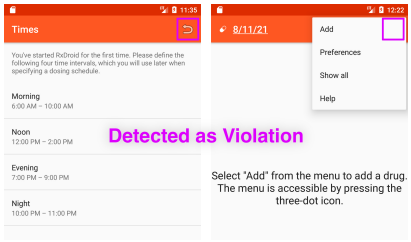

Figure 8: Persisting Elements Detection

guideline and could not be closed by a tappable element. We then reported the metrics seen in Table 5

5 EMPIRICAL RESULTS

5.1 RQ$_1$: Expanding Section Detector Accuracy?

The expanding section detector performed well across nearly all of our studied metrics, as indicated in Table 5. With a precision of .9042, F1-Score of 0.9129, and an accuracy of 0.9123, this detector shows promising results that it is capable of identifying a section's "collapsibility" accurately. This indicates that, by using MotorEase, developers will have an increased chance of identifying sections that were designed without a method of closure which may impede motor-impaired users. Importantly, as per Table 5 our approach surpassed the performance of Groundhog's ability to detect closable sections, with a precision of 0.6849, demonstrating its effectiveness in comparison. The difference between the two is attributable to the fact that MotorEase and Groundhog do not have the same objective. MotorEase aims to detect the presence or absence of closing icons, while Groundhog aims to determine whether a closing icon can be accessed using an assistive service. If a closing icon is missing, Groundhog cannot detect it. Since there is no closing icon, Groundhog does not even attempt to close it using an assistive service. Groundhog relies on an accessibility service to detect icons on the screen, However, if the icon does not have any metadata indicating that it is an interactive, it is unable to interact with the icon and close it. MotorEase's novelty lies in its ability to consider the visual presence of the icon independent of its metadata description. However, MotorEase's detector does struggle to detect certain instances in the MotorEase benchmark. An example of a successful and unsuccessful prediction is shown in Figure 7. The example on the left side is an expanding section that the detector correctly identified as collapsible. It is correctly identified by MotorEase as a closing section because of the semantic matching's ability to generalize the word "cancel" as a means of closing the section. The image on the right is not collapsible but the detector deduced that it was collapsible. This was due to the use of X icons in the training data for the object detection model. The "X" icons used in this image imply the deletion of text, but this detectors object detection model is also trained to identify "X" icons that may be used to close the expanding section. It should be noted that this was an outlier in our dataset, and that the pattern for detecting "X" icons generally worked as expected.

5.2 RQ$_2$: Visual Touch Target Detector Accuracy?

The touch-target detector performed well as illustrated in Table 5. The detector exhibited perfect precision, an F1-Score of 0.7986, and an accuracy of 0.8525, showing encouraging results of its ability to detect and classify screens with touch target violations well. Given the results, this detector is successfully able to give developers an insight into smaller icons that may inconvenience users with tremors and inaccurate touches. Importantly, our approach surpassed Google Accessibility Scanners's ability to detect visually small elements on the screen, which only had a precision of 0.5085, demonstrating its effectiveness in comparison. The primary reason for the gap in performance is that Google Accessibility Scanner is not able to check the *visual* touch-target size, and can only parse the reported size from the `uiautomator` framework, which may not necessarily correspond to the visual touch target size. However, while nearly all violations returned by this detector are correct, it does tend miss certain types of violations. For instance, it cannot detect icons on the screen which are not labeled as clickable in the XML. By default, all elements on an android device have a clickable property with a boolean True/False label. If that property is labeled as False, MotorEase does not consider the object to be clickable. Dynamically generated screenshots may not always have the metadata information for each element on the screen, and this absence of information was the main reason for incorrect or missed violations. This detector could be augmented in the future with work from the HCI community aimed at assessing icon tap-ability [87].

5.3 RQ$_3$: Persisting Element Detector Accuracy?

This results of this detector are presented in Table 5. With a precision of 0.8214, F1-Score of 0.8846, and accuracy of 0.8786, this detector shows encouraging signs of viability. This detector's recall rate of 0.9583 suggests that the detector identifies true positives at

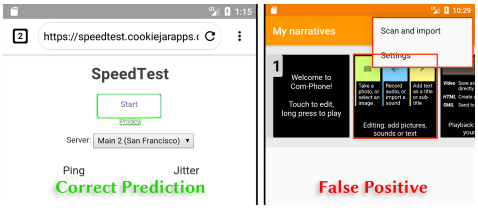

Figure 9: Visual Icon Distance Detection

a high rate. This detector, however, relies heavily on the XML to locate elements on the screen, which can lead to mis-classifications. One such example is shown in Figure 8. The example shows the undo icon on the screenshot on the left and a menu on the right side where the undo icon would be. The XML for this second screen has the undo icon in the data, but its bounds and information are missing since they are not visible on the screen. This was classified as a violation though it is not a violation.

5.4 RQ$_4$: Visual Icon Distance Detector Accuracy?

This results of this detector are presented in Table 5. With a precision of 0.7119, F1-Score of 0.8317, and accuracy of 0.9575. This detector achieved a perfect recall rate, suggesting that this detector provides developers with a reliable tool that is capable of accurately detecting closely placed icons, prompting potential UI design and icon placement adjustments. Like the Visual Touch-Target Violation detector, this detector relies heavily on the `uiautomator` metadata specifying clickable components, and the accuracy of the UIED element bound detector. The latter led to certain cases of inaccurate reporting of violations, due to incorrect overlapping bounds. Both of these examples are shown in Figure 9. The first example shows a correct prediction where MotorEase correctly identifies two icons on the screen and determines that they are not a minimum of 8 pixels in distance. However, Figure 9 also has an example of a false positive prediction which shows an overlap of two elements on the screen. Due to the visual bounds of each overlapping, the distance between the two is 0, therefore predicting a false positive.

5.5 RQ$_5$: False Positives and Negatives

The confusion matrices for the detectors are shown in Figure 10, where green boxes illustrate predictions that matched the ground truth, and red boxes illustrate predictions that did not match the ground-truth. These figures provide a visual representation of false positive and negative rates. Figure 10-② illustrates that the visual touch target detector never produced a false-positive outcome. This is due to the fact that the detector specifically extracts elements labeled as clickable in the XML, therefore once they are extracted and have their edges analyzed, the detector is able to detect violations with certainty. The confusion matrix for the expanding section detector is shown in Figure 10-①, there were 23 false positive predictions, mainly due to limitations related to lexical pattern matching. Finally, MotorEase's persisting element detector identified only 8 false positives, mainly due to inconsistencies in matching elements across screens due to unexpected changes in `uiautomator` XML files. In evaluating the effectiveness of MotorEase, we also

① **Expanding Section Closure Confusion Matrix**

	Ground Truth		
Predicted	Positive	Negative	Total
Positive	220	23	223
Negative	19	217	236
Total	239	240	

② **Visual Touch-Target Size Confusion Matrix**

	Ground Truth		
Predicted	Positive	Negative	Total
Positive	117	0	117
Negative	59	224	283
Total	176	224	

③ **Persisting Element Location Confusion Matrix**

	Ground Truth		
Predicted	Positive	Negative	Total
Positive	23	5	28
Negative	1	20	21
Total	24	25	

④ **Adjacent Visual Icon Distance Confusion Matrix**

	Ground Truth		
Predicted	Positive	Negative	Total
Positive	42	17	59
Negative	0	341	341
Total	42	258	

Figure 10: Detector Confusion Matrices

considered the impact of false negative predictions, as they signal violations that are not flagged by MotorEase, and hence could reach end-users. However, our evaluation revealed that the recall rate of MotorEase was relatively high, indicating that it is a dependable tool with a high true positive rate, thus demonstrating its practical applicability. In regards to MotorEase's low false negative prediction rate, The confusion matrix analysis for the icon distance detector shown in Figure 10-④ shows that MotorEase predicted 0 false negatives. Moreover, the confusion matrix analysis for the persisting elements detector shown in Figure 10-③ showed only one false negative prediction. Similarly, the confusion matrix for the expanding section detector presented in Figure 10-① demonstrated a false negative rate similar to its false positive rate, further supporting the viability of MotorEase. Overall, our evaluation demonstrates that MotorEase is likely a generally practical tool, exhibiting a relatively low rate of false positives and negatives. MotorEase leverages it's ability to visually comprehend the visual and textual contents of a screen to determine accessibility violations. This approach offers distinct advantages. For instance, MotorEase showcases its adaptability by effectively detecting accessibility guideline violations regardless of variations in UI design or format. In contrast, traditional metadata-based approaches might be limited in their ability to analyze physical elements on the screen.

6 RELATED WORK

6.1 Accessibility Studies on Mobile Apps

There exists a large body of work that aims to understand how users with disabilities use their devices, and the potential accessibility issues that exist in current software applications [21, 22, 36, 58, 59, 63, 71, 82, 85, 90]. Such studies tend to take two forms, user studies [21, 58] and empirical analyses of software [36, 90].

A study by Alshayban et al. [23] examined the current state of accessibility issues in Android applications [23] using a Google-provided accessibility testing framework [13]. They found that Text Contrast, Touch Target, Image Contrast, and Speakable Text are the most frequent accessibility issues [23]. Vendome et al. [88] also examined the prevalence of accessibility issues in Android apps. In this study, the authors mined thousands of android applications and analyzed the usage of accessibility APIs and whether or not applications adhered to accepted guidelines. In addition, they mined thousands of messages on Stack Overflow and other interaction platforms to understand the sentiment of developers and the types of questions they were asking. They found that most accessibility

based conversations were centered around LV features while a lesser number focused on DHH features.

Our approach is motivated by and complements what researchers have discovered in the above studies. These studies have shown the importance of the problem that MotorEase tackles through illustrating the widespread prevalence of accessibility issues in mobile apps and illustrating a comparative lack of awareness of motor-impairment design guidelines and the needs of such users. Hence, these studies both motivate and validate our work on MotorEase.

6.2 Accessibility Testing

Software testing for accessibility aids developers in identifying violations of guidelines set forth by companies and governments [1, 13, 69, 73]. A wide range of research has been carried out to automate this process [25, 29, 38, 69, 77, 80, 81]. We discuss the most closely related approaches below.

Eler et.al. introduced the MATE tool [38] that uses automated dynamic analysis to check for accessibility issues that affect users with visual impairments in mobile apps, and generate detailed reports that facilitate developers fixing identified issues. Similar to MATE, MotorEase also leverages automated dynamic analysis and is able to generate reports that aid developers in fixing accessibility issues. However, our tool is differentiated by the ML-based analyses employed, and by it's focus on motor-impaired users.

Latte [80] is an accessibility testing framework introduced by Salehnamadi et.al. for android applications that aims to provide a deeper analysis compared to testing frameworks provided by Google [9, 13] by testing how accessibility services, such as VoiceOver, function in conjunction with feature-based use cases. The authors carried out an evaluation of their tool using the `switchAccess` and `TalkBack` services [1, 13] and found that several applications did not accommodate for both forms of accessible interactions. Latte is one of the only tools or studies that explicitly considers accessibility issues for users with motor impairments, given that it is capable of integrating with the `switchAccess` service in Android. However, given Latte's use case driven nature it both (1) requires pre-existing test cases, which many mobile apps have been shown to lack [49], and (ii) cannot detect violations of the specific accessibility guidelines targeted by MotorEase, as it attempts to carry out actions of a use case using an assistive service, and does not analyze the UI for specific patterns. In short, MotorEase and Latte serve largely *complementary* purposes, that is MotorEase provides UI design guidance to developers to avoid common pitfalls related to motor-impaired accessibility issues, and Latte can point out issues specific to given use cases and accessibility services.

Chen et al. [33] introduced Xbot, an accessibility testing tool that is capable of identifying accessibility issues within an app using a combination of dynamic and static program analysis. Xbot is not able to uncover any of the accessibility issues targeted by MotorEase. MotorEase exhibits novelty as compared to Xbot as it utilizes semantic understanding of the visual and textual elements of UI screens to detect new issues that affect motor-impaired users.

Finally, recently Salehnamadi et.al introduced the Groundhog approach [81], which is an accessibility crawler for mobile apps. Groundhog implements an automated UI crawler that explores an app both with and without assistive services, such as `TalkBack`, and notes any cases where an action can be performed through traditional touches, but cannot be performed via an assistive service. Again, MotorEase is largely *complementary* to this work, in that Groundhog targets general issues related to actionability and locatability of UI elements more broadly, but does not target the specific motor-impairment accessibility issues addressed by MotorEase. In fact, MotorEase aims to address two specific classes of issues identified in the Groundhog paper as being important for future work, (i) counterintuitive navigation (e.g., persisting elements) and (ii) inoperative actions (e.g., expanding sections).

6.3 Accessibility-Based UI Comprehension

Given that users interact with software through a UI, and past work has illustrated accessibility issues present in UIs [26, 47, 75, 80], there is a body of work dedicated to automatically comprehending and augmenting UIs to identify and circumvent accessibility and UI issues [31, 40, 53, 54, 62, 64, 67, 84, 91].

UI elements and icons typically need to be labeled in order for screen readers to be able to properly describe their appearance and functionality, however, such metadata is often missing from apps [32]. Zhang et.al.[91] and Chen et.al. [32] designed machine learning models trained from both existing UI labels and annotated labels from developers. Follow-up work by Mehralian et. al. introduced COALA [60], which aimed to improve upon the automated icon labeling by considering context related to screen text and region to build a multi-modal model for predicting icon labels.

Mansur et.al. introduced the AidUI [56] tool, which uses semantic screen understanding to automatically identify and localize Dark Patterns in mobile app and web user interfaces. This technique uses similar techniques for semantic screen understanding as MotorEase, but in different ways. For instance, while AidUI uses element size and distance between elements as a factor in determining certain dark patterns, MotorEase aims to compare to the programmatic element size and visual element size to detect accessibility violations. Further, MotorEase analyzes *both* visual UI screenshots and `uiautomator` metadata, whereas AidUI operates only upon visual UI screenshots – with the use of multimodal data serving as a source of novelty for MotorEase.

7 CONCLUSION & FUTURE WORK

In this paper, we presented MotorEase, an approach for detecting, classifying, reporting motor-impairment accessibility violations. We measured the performance, generalizability, and applicability of MotorEase to various open source applications. Our results indicate that MotorEase is effective in practice and offers a novel approach for developers to identify accessibility issues affecting motor-impaired users. Future work will examine the potential to detect accessibility issues in web apps and conduct user studies.

ACKNOWLEDGEMENTS

This work is supported in part by NSF grants CCF-1955853 and CNS-2132285 and a gift from Dragon Testing Ltd. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors. We would also like to express our sincere thanks to the authors of the Groundhog tool for their assistance in running the tool on the MotorCheck benchmark.

REFERENCES

[1] Accessibility: Apple human interface guidelines. https://developer.apple.com/design/human-interface-guidelines/foundations/accessibility/.

[2] Accessibility scanner. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor.

[3] Android switch access service. https://support.google.com/accessibility/android/answer/6122836?hl=en.

[4] Google cloud vision api. https://cloud.google.com/vision/docs/ocr.

[5] Motorease github repository. https://github.com/SageSELab/MotorEase.

[6] Motorease website. https://sagelab.io/MotorEase.

[7] Motorease zenodo archive. https://zenodo.org/doi/10.5281/zenodo.10460700.

[8] Universal design definition and guidelines. https://www.section508.gov/blog/Universal-Design-What-is-it/.

[9] "design for android: android developers," https://developer.android.com/design.

[10] Ada web accessibility lawsuit recap report https://blog.usablenet.com/2018-ada-web-accessibility-lawsuit-recap-report, 2018.

[11] Adalaws https://www.ada.gov/cguide.htm, 2019.

[12] Accessibility guide https://accessibility.18f.gov/checklist/accessibility, 2022.

[13] Accessibility https://developer.android.com/guide/topics/ui/accessibility, 2022.

[14] Motor impairment https://accessibility.huit.harvard.edu/disabilities/motor-impairment?page=1, 2022.

[15] Torchvision- torchvision main documentation. https://pytorch.org/vision/stable/index.html, 2022.

[16] Web content accessibility guidelines (wcag). https://www.w3.org/TR/WCAG21/, journal=W3C, 2022.

[17] https://f-droid.org/, F-droid.

[18] https://play.google.com/store, Google Play Store.

[19] https://labelstud.io, LabelStudio.

[20] J. Abascal, A. Aizpurua, I. Cearreta, B. Gamecho, N. Garay-Vitoria, and R. Miñón. Automatically generating tailored accessible user interfaces for ubiquitous services. In *The Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '11, page 187–194, New York, NY, USA, 2011. ACM.

[21] A. Aizpurua, M. Arrue, S. Harper, and M. Vigo. Are users the gold standard for accessibility evaluation? In *Proceedings of the 11th Web for All Conference*, W4A '14, New York, NY, USA, 2014. ACM.

[22] L. D. A. Almeida and M. C. C. Baranauskas. Universal design principles combined with web accessibility guidelines: A case study. In *Proceedings of the IX Symposium on Human Factors in Computing Systems*, IHC '10, page 169–178, Porto Alegre, BRA, 2010. Brazilian Computer Society.

[23] A. Alshayban, I. Ahmed, and S. Malek. Accessibility issues in android apps: State of affairs, sentiments, and ways forward. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1323–1334, New York, NY, USA, 2020. ACM.

[24] D. Astler, H. Chau, K. Hsu, A. Hua, A. Kannan, L. Lei, M. Nathanson, E. Paryavi, M. Rosen, H. Unno, C. Wang, K. Zaidi, X. Zhang, and C.-M. Tang. Increased accessibility to nonverbal communication through facial and expression recognition technologies for blind/visually impaired subjects. In *The Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '11, page 259–260, New York, NY, USA, 2011. ACM.

[25] M. Bajammal and A. Mesbah. Semantic web accessibility testing via hierarchical visual analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1610–1621, 2021.

[26] M. Bajammal and A. Mesbah. Semantic web accessibility testing via hierarchical visual analysis. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1610–1621. IEEE Press, 2021.

[27] C. Bernal-Cardenas, N. Cooper, M. Havranek, K. Moran, O. Chaparro, D. Poshyvanyk, and A. Marcus. Translating video recordings of complex mobile app ui gestures into replayable scenarios. *IEEE Transactions on Software Engineering*, 49(04):1782–1803, apr 2023.

[28] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 309–321, New York, NY, USA, 2020. ACM.

[29] G. Brajnik, C. Pighin, and S. Fabbro. Model-based automated accessibility testing. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '15. ACM, 2015.

[30] R. Calvo, F. Seyedarabi, and A. Savva. Beyond web content accessibility guidelines: Expert accessibility reviews. In *Proceedings of the 7th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-Exclusion*, DSAI 2016, page 77–84, New York, NY, USA, 2016. ACM.

[31] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 665–676, New York, NY, USA, 2018. ACM.

[32] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, G. Li, and J. Wang. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 322–334, New York, NY, USA, 2020. ACM.

[33] S. Chen, C. Chen, L. Fan, M. Fan, X. Zhan, and Y. Liu. Accessible or not? an empirical investigation of android app accessibility. *IEEE Transactions on Software Engineering*, 48(10):3954–3968, 2022.

[34] P. T. Chiou, A. S. Alotaibi, and W. G. Halfond. Bagel: An approach to automatically detect navigation-based web accessibility barriers for keyboard users. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. ACM.

[35] P. T. Chiou, A. S. Alotaibi, and W. G. J. Halfond. Detecting and localizing keyboard accessibility failures in web applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 855–867, New York, NY, USA, 2021. ACM.

[36] H. N. da Silva, A. T. Endo, M. M. Eler, S. R. Vergilio, and V. H. S. Durelli. On the relation between code elements and accessibility issues in android apps. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, SAST 20, page 40–49, New York, NY, USA, 2020. ACM.

[37] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, page 845–854, New York, NY, USA, 2017. ACM.

[38] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 116–126, 2018.

[39] L. Flórez-Aristizábal, S. Cano, C. A. Collazos, A. F. Solano, and S. Brewster. DesignABILITY. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, May 2019.

[40] K. Z. Gajos, J. O. Wobbrock, and D. S. Weld. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, page 231–240, New York, NY, USA, 2007. ACM.

[41] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su. Practical gui testing of android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 269–280. IEEE Press, 2019.

[42] M. Havranek, C. Bernal-Cárdenas, N. Cooper, O. Chaparro, D. Poshyvanyk, and K. Moran. V2s: A tool for translating video recordings of mobile app usages into replayable scenarios. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 65–68, 2021.

[43] S. K. Kane, M. R. Morris, A. Z. Perkins, D. Wigdor, R. E. Ladner, and J. O. Wobbrock. Access overlays. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, Oct. 2011.

[44] B. A. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 07 2007.

[45] J. Kong, M. Zhong, J. Fogarty, and J. O. Wobbrock. New metrics for understanding touch by people with and without limited fine motor function. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '21, New York, NY, USA, 2021. ACM.

[46] J. Li, G. W. Tigwell, and K. Shinohara. Accessibility of high-fidelity prototyping tools. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, May 2021.

[47] J. Li, Z. Yan, E. H. Jarjue, A. Shetty, and H. Peng. Tangiblegrid: Tangible web layout design for blind users. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, UIST '22, New York, NY, USA, 2022. ACM.

[48] Y. Li, Z. Yang, Y. Guo, and X. Chen. Droidbot: a lightweight ui-guided test input generator for android. In *ICSE-C*. IEEE, 2017.

[49] J.-W. Lin, N. Salehnamadi, and S. Malek. Test automation in open-source android apps: A large-scale empirical study. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1078–1089, New York, NY, USA, 2021. ACM.

[50] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 111–122. IEEE Press, 2015.

[51] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. How do developers test android applications? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 613–622, 2017.

[52] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410, 2017.

[53] Z. Liu. Discovering ui display issues with visual understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1373–1375, New York, NY, USA, 2021. ACM.

[54] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang. Owl eyes: Spotting ui display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 398–409, New York, NY, USA, 2021. ACM.

[55] I. S. MacKenzie, R. W. Soukoreff, and J. Helga. 1 thumb, 4 buttons, 20 words per minute: Design and evaluation of h4-writer. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 471–480, New York, NY, USA, 2011. ACM.

[56] S. H. Mansur, S. Salma, D. Awofisayo, and K. Moran. Aidui: Toward automated recognition of dark patterns in user interfaces. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1958–1970. IEEE, 2023.

[57] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *ISSTA*. ACM, 2016.

[58] D. A. Mateus, C. A. Silva, M. M. Eler, and A. P. Freire. Accessibility of mobile applications: Evaluation by users with visual impairment and by automated tools. In *Proceedings of the 19th Brazilian Symposium on Human Factors in Computing Systems*, IHC '20, New York, NY, USA, 2020. ACM.

[59] T. B. McHugh and C. Barth. Assistive technology design as a computer science learning experience. In *Proceedings of the 22nd International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '20, New York, NY, USA, 2020. ACM.

[60] F. Mehralian, N. Salehnamadi, and S. Malek. Data-driven accessibility repair revisited: On the effectiveness of generating labels for icons in android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 107–118, New York, NY, USA, 2021. ACM.

[61] L. R. Milne and R. E. Ladner. Blocks4all. In *CHI'18*. ACM, Apr. 2018.

[62] K. Montague, V. L. Hanson, and A. Cobley. Designing for individuals: Usable touch-screen interaction through shared user models. In *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '12, page 151–158, New York, NY, USA, 2012. ACM.

[63] K. Montague, H. Nicolau, and V. L. Hanson. Motor-impaired touchscreen interactions in the wild. In *Proceedings of the 16th International ACM SIGACCESS Conference on Computers ; Accessibility*, ASSETS '14, page 123–130, New York, NY, USA, 2014. ACM.

[64] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk. Automated reporting of gui design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 165–175, New York, NY, USA, 2018. ACM.

[65] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk. Crashscope: A practical tool for automated testing of android applications.

[66] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 33–44, 2016.

[67] K. Moran, C. Watson, J. Hoskins, G. Purnell, and D. Poshyvanyk. Detecting and summarizing gui changes in evolving mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 543–553, New York, NY, USA, 2018. ACM.

[68] MulongXie. Uied - ui element detection, detecting ui elements from ui screenshots or drawnings, 2021.

[69] K. Norman, Y. Arber, and R. Kuber. How accessible is the process of web interface design? In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, New York, NY, USA, 2013. ACM.

[70] F. Nunes, P. A. Silva, J. Cevada, A. C. Barros, and L. Teixeira. User interface design guidelines for smartphone applications for people with parkinson's disease. *Universal Access in the Information Society*, 15(4):659–679, Oct. 2015.

[71] U. Oh, S. K. Kane, and L. Findlater. Follow that sound: Using sonification and corrective verbal feedback to teach touchscreen gestures. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, New York, NY, USA, 2013. ACM.

[72] P. Parhi, A. K. Karlson, and B. B. Bederson. Target size study for one-handed thumb use on small touchscreen devices. In *Proceedings of the 8th Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '06, page 203–210, New York, NY, USA, 2006. ACM.

[73] K. Park, T. Goh, and H.-J. So. Toward accessible mobile application design: Developing mobile application accessibility guidelines for people with visual impairment. In *Proceedings of HCI Korea*, HCIK '15, page 31–38, Seoul, KOR, 2014. Hanbit Media, Inc.

[74] A. Pavel, G. Reyes, and J. P. Bigham. Rescribe. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, Oct. 2020.

[75] Y.-H. Peng, M.-T. Lin, Y. Chen, T. Chen, P. S. Ku, P. Taele, C. G. Lim, and M. Y. Chen. Personaltouch: Improving touchscreen usability by personalizing accessibility settings based on individual user's touchscreen interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–11, New York, NY, USA, 2019. ACM.

[76] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In A. Moschitti, B. Pang, and W. Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[77] N. P. K. Ramachandra and C. Csallner. Testing web-based applications with the <u>v</u>oice <u>c</u>ontrolled <u>a</u>ccessibility and <u>t</u>esting tool (vcat). In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 208–209, New York, NY, USA, 2018. ACM.

[78] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[79] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock. Examining image-based button labeling for accessibility in android apps through large-scale analysis. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. ACM, Oct. 2018.

[80] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. ACM.

[81] N. Salehnamadi, F. Mehralian, and S. Malek. Groundhog: An automated accessibility crawler for mobile apps. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, 2022.

[82] M. T. Santiago and A. B. Marques. Are user reviews useful for identifying accessibility issues that autistic users face? an exploratory study. In *Proceedings of the 21st Brazilian Symposium on Human Factors in Computing Systems*, IHC '22, New York, NY, USA, 2022. ACM.

[83] Z. Sarsenbayeva, N. van Berkel, E. Velloso, J. Goncalves, and V. Kostakos. Methodological standards in accessibility research on motor impairments: A survey. *ACM Comput. Surv.*, may 2022. Just Accepted.

[84] B. N. Shiver and R. J. Wolfe. Evaluating alternatives for better deaf accessibility to selected web-based multimedia. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '15, page 231–238, New York, NY, USA, 2015. ACM.

[85] G. M. S. Silva, R. M. de C. Andrade, and T. de Gois R. Darin. Design and evaluation of mobile applications for people with visual impairments: A compilation of usable accessibility guidelines. In *Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems*, IHC '19, New York, NY, USA, 2019. ACM.

[86] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 245–256, New York, NY, USA, 2017. ACM.

[87] A. Swearngin and Y. Li. Modeling mobile interface tappability using crowdsourcing and deep learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–11, New York, NY, USA, 2019. ACM.

[88] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez. Can everyone use my app? an empirical study on accessibility in android apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–52, 2019.

[89] W. Wang, W. Yang, T. Xu, and T. Xie. Vet: identifying and avoiding UI exploration tarpits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 83–94, Athens Greece, Aug. 2021. ACM.

[90] S. Yan and P. G. Ramachandran. The current status of accessibility in mobile apps. *ACM Trans. Access. Comput.*, 12(1), feb 2019.

[91] X. Zhang, L. de Greef, A. Swearngin, S. White, K. Murray, L. Yu, Q. Shan, J. Nichols, J. Wu, C. Fleizach, A. Everitt, and J. P. Bigham. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. ACM.

[92] X. C. Zhang, K. Fang, and G. Francis. Optimization of switch keyboards. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '13, New York, NY, USA, 2013. ACM.

[93] Y. Zhao, S. Talebipour, K. Baral, H. Park, L. Yee, S. A. Khan, Y. Brun, N. Medvidović, and K. Moran. Avgust: automating usage-based test generation from videos of app executions. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 421–433, New York, NY, USA, 2022. ACM.

[94] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5551–5560, 2017.