Computational Understanding of User Interfaces

Jason Wu

June 2024

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jeffrey P. Bigham, Chair
Jeffrey Nichols
Jodi Forlizzi
Sherry Tongshuang Wu
Tom Mitchell

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Keywords: Accessibility, Computational Modeling, Human-Computer Interaction, User Interface

TBD

Abstract

A grand challenge in human-computer interaction (HCI) is constructing user interfaces (UIs) that make computers useful for all users across all contexts. Conventional UI development processes have approached this goal by iteratively converging towards a single "final" UI through prototyping, implementation, and testing. However, even when following best practices, this approach locks in a set of assumptions that often cannot accommodate the diversity of user abilities, usage contexts, or computing technologies, ultimately limiting how we can use computers. For example, UIs designed for one context might not perform well in another, and UIs can be inaccessible to users who different abilities and preferences. In this dissertation, I propose a new approach that uses machine-learning-driven systems that automatically understand and manipulate existing UIs. Using content and functionality inferred from the UI, combined with sensed usage context, a new interface can be synthesized that better meets the immediate needs of individual users.

My work represents the initial technical foundation for this vision. First, I describe approaches for understanding user ability and context (*user understanding*), which HCI suggests is the basis for building good interfaces. I describe a recommendation system that recommends device settings (*e.g.,* accessibility features) based on sensed usage behaviors and user interaction logs. Results from a user study showed that the majority (74%) of predicted recommendations were rated as helpful. Nevertheless, this approach of adapting interfaces through configuration changes has traditionally been limited, since applications often do not properly expose their semantics to external services. To this end, I describe several projects in the area of *UI understanding*, which shows that it is possible to overcome this barrier using data-driven ML models that predict interface layout, structure, and functionality from visual information, which is how UIs are generally assumed to be used. These predicted semantics can enable many forms existing computing infrastructure, such as accessibility and UI agents to work more reliably and robustly. Finally, I combine both user and UI understanding to *dynamically generate and adapt UIs* that meet the specific needs of users. I describe ML-driven systems that generate UIs by modifying existing application layouts and generating UI code based on personalized user profiles and design objectives. Ultimately, through my work, I show that computational understanding of user interfaces allows UIs to be transformed from static objects into malleable representations that can be dynamically reshaped for new devices, modalities, and users.

Acknowledgments

TBD

Contents

List of Figures

List of Tables

1 Introduction

What are computers going to look like in the future? How are we going to interact with them? Right now is an exciting time to be thinking about this question. There are so many visions imagining how computers can augment reality, respond naturally to our queries, and even how they can understand more implicit interactions like movements and gestures. The key promise here is, by doing this, computers can be more natural to use, more personalized, more adaptable, and more context-aware. This will allow users the flexibility interact with digital information whoever they are and wherever they are.

But if we look at user interfaces (UIs) today, we realize that they are not quite ready yet. A lot of the functionality that we use computers for are locked behind UIs built for a rigid set of assumptions. For example, many UIs are designed to run only on one specific device and are pre-programmed for a specific layout. Even running applications on devices with even slightly different display characteristics can substantially impact usability, and larger discrepancies between designed and observed context can lead to even more significant degradation. Currently, UIs also make a lot of assumptions about users. Because of this many non-experts find applications hard to use or hard to learn. UIs are often not created accessibility – what if a user is blind and can't see the screen, or has some other accessibility needs? Even when we've developed other programs like assistive technology and development tools to address some of these problems, we find that they often don't work with UIs that are missing the required APIs and toolkit support. These are just a few examples of the assumptions that are baked into UIs today that limit how we can use computers.

A grand challenge in the field of human-computer interaction (HCI), is overcoming these challenges, and answering the question: How do we make these computer interfaces useful for all people, across all contexts? One reason this is hard is because today, when most UIs are manually designed and compiled into an application, there's no good way for us to dynamically change them. Conventional UI design processes [55] often advocate for identifying and resolving these all possible requirements at design-time, guided by an iterative process of sketching, prototyping, and user testing. The ultimate goal of this idealized process is to converge on a single interface or other type of "deployed" artifact that is expected to be interacted with in pre-verified ways. However, this is exceedingly difficult and time-consuming to manually design a single "optimal" UI because of the trade-offs that go into designing for different scenarios and the mismatches between expected and actual usage conditions.

In this dissertation, I investigate a new approach that shifts much of the burden away from designers and developers to machine-learning-driven systems that can automatically understand and manipulate existing UIs the way that people are assumed to be able to do, typically through visual perception and touch interaction. Using content and functionality extracted from the UI, combined with sensed usage context, a new interface can be synthesized that better meets the immediate needs of the user.

> *Computational understanding of user interfaces* allows UIs to be transformed from static objects into malleable representations that can be dynamically reshaped for new devices, modalities, and users.

1.1 Document Organization

This dissertation presents several projects that advance this vision of using computational understanding and improving UIs. In Chapter 2, I contextualize my work by reviewing relevant prior work, focusing on existing approaches for addressing this problem through design, adaptation, and generation. I present three areas of work that serve as the technical foundation for my approach.

First, I investigate *user understanding* — how can machines understand user needs and context, which HCI suggests is the basis for building good interfaces? I focus specifically on ML-driven approaches to do this automatically, by implicitly learning it from usage behavior and interactions. Chapter 3, presents a system that recommends existing OS features and settings as a mechanism for transforming the appearance and behavior of existing apps based on observed usage behavior. An evaluation of this system showed that the majority of recommendations produced by this system were found to be useful by users. However, even when the correct settings were enabled, a lot of applications did not properly support them, due to the lack of required metadata or missing APIs.

The next part of my dissertation focuses on *UI understanding*, which aims to address the question — how can machines reliably understand and operate user interfaces? In Chapters 4, 5, and 6, I introduce machine learning-driven systems that predict this semantic information from and operate existing apps purely from visual appearance (*i.e.,* screenshot).

Finally, I combine computational understanding of user context and UIs to improve experience through *UI generation*. Chapter 7 presents a system that optimizes existing app layouts based on user calibration data to improve target acquisition time. Chapter 8 and 9 focus on UI code generation as a method of enabling high-quality UIs to be dynamically synthesized to meet user needs. I present projects that ensure UI code generated by generative models are syntactically correct and well-designed.

Chapter 10 concludes this dissertation by discussing the contributions and impact of my work, and opportunities for future research.

2

Related Work

2.1 Prior Approaches to UI Adaptation

2.1.1 Strategies During Development

Adaptive User Interfaces

Adaptive user interfaces (AUIs) can improve user experience by automatically adapting how information and functionality are presented in a user interface. However, the dynamic nature and potentially numerous variations of AUIs make them challenging to author. A range of methods has been employed to define adaptive behavior. One approach is to develop tools and frameworks that contain pre-programmed logic for common adaptive patterns. For example, many commercial tools (e.g., Dreamweaver[24], WebFlow [22]) and frameworks (e.g., Bootstrap [21]) contain ready-made templates for adapting a website to mobile and tablet form-factors. Previous work has developed similar software frameworks to support context-awareness [85, 208], user behavior [119], and mixed-reality [166] UIs by providing developers with pre-built, composable modules for sensing, recognition, and adaptation. Model-based approaches have also been developed, which provide higher-level abstractions of widgets and behavioral patterns common in ubiquitous [255] and multi-device computing applications [231].

Other development approaches have used objective-based optimization to automatically adapt UIs by searching for layouts that optimize predefined metrics [99, 110, 234, 252]. The ARNAULD and SUPPLE [110] systems, enable developers to specify a high-level, formal definition of their UI that is used for dynamic generation. During runtime, these systems use an optimization algorithm that makes rendering decisions based on an objective function that captures user preferences or abilities. Similarly, recent approaches for the 3D placement of UI widgets in mixed reality applications applied optimized layouts based on the semantic properties of UIs [68, 210] and developer-tuned objective functions [95] such as reachability, visibility, and consistency.

Ability-based Design

Ability-based design is a design process that focuses on the ability of users during the development process which allows better experience for diverse users [322]. The simplest but perhaps most effective approach is to create a new UI for each user group. Simple Finder is an alternative app to navigate the Mac OS filesystem that was designed for young kids and elderly, which is easier to learn due to its distilled functionality. However, even with best development practices and decoupling, implementing this may require a lot of effort. There has been research in automating certain aspects of ability-based design. A well-known example is SUPPLE, which allows developers to define certain aspects of the user interface and run an optimization algorithm to personalize the resulting interface to facilitate faster access and lower error [115]. This approach has shown encouraging results for optimizing application mockups for low-vision and motor impaired users [114]. Adaptive UIs have also been constructed to provide accessibility benefits to older adults with motor and cognitive impairments [273]. However, often this approach to developing application requires instrumentation of the code and still places the burden of adaptation on the developer.

2.1.2 Strategies During Runtime

Matching People to Access Technologies

Traditionally, matching people to accessible technologies has been done by human experts or through recommendations of medical professionals. Assistive Technology (AT) specialists can be employed as consultants to provide guidance on making content (*e.g.,* educational curriculum) accessible [1]. Physicians or therapists may provide guidance on using assistive technology as part of rehabilitation therapy [6, 7], although this sort of support is not available to everyone who could benefit from accessibility features. Such matching is typically done in specialized environments and is costly in terms of time and money. Access technologies tend to have low adoption rates [76], perhaps because potential users do not have sufficient time to see how they would work into their daily lives. In contrast to this matching process, many people who could benefit from accessibility features on mobile devices they may already own may not know to seek external help or have access to it.

Interface Repair

Adaptive UIs that need to be built from scratch, using specific UI toolkits, and with specific requirements on developers, is severely limiting in practice. It is difficult to get developers to adopt any new UI framework, and this approach does not address the substantial legacy of existing applications not created to support this use. An alternative is to repurpose and augment existing applications. For example, existing UIs can be retargeted to support new modalities [296] or support responsive resizing [157, 326]. *Interaction proxies* apply runtime modification to existing mobile apps by “repairing” inaccessible or difficult-to-use UI elements [345]. Interaction proxies and other approaches that apply input/output redirection [292] use specialized UI elements to re-render parts of the original UI while maintaining interactivity.

To support a broad range of existing applications by not having to rely on application dependencies or other application metadata, some approaches improve applications only from their pixels (visual appearance). Template-based pixel matching has been used to locate icons or UI elements of interest on a screen to support early screen readers [279] and end-user scripting [339]. Similarly, Prefab enables custom interaction techniques (*e.g.*, target-aware pointing) for desktop applications through pixel-based identification of user interface elements [87].

2.2 Computational UI Understanding

2.2.1 User Understanding

Automated Screening

Recent work has started to explore how to “detect” whether someone is likely to have a particular condition, which could be used to recommend that they consult an expert or even try out a particular assistive technology.

Many mobile health sensing efforts have focused on providing low-cost alternatives for monitoring chronic symptoms, such as asthma [175] and cystic fibrosis [126], or building mobile “screening” applications for detecting medical conditions [78, 313, 316]. Some screening applications reduce the dependence on a specialized procedure and involve completing a task, such as playing a game, or leveraging interaction data with specialized apps (*e.g.,* photo browser, lock screen) [265, 266, 304]. However, many of these approaches require active intervention on the user’s part (*e.g.,* opening an app and performing a specialized screening procedure); as we later show, most people would not think to check their mobile device for accessibility affordances. This suggests the need to passively detecting accessibility needs by monitoring natural interactions with unmodified applications. Downloading and using a screening app would indicate some knowledge of accessibility features.

Modeling Touch Interaction

Touch interaction can be analyzed using existing models of cursor-based selection [219], which has been extended to the touch screen [42]. This type of analysis gives recommendations about the relative size and spacing of UI elements on touch-based apps. Other work has focused on properties specific to touch interaction, such as the effect of the finger choice on selection accuracy. As examples, touch input is imprecise due to factors such as finger deformations (i.e., the “fat finger” problem [312]), occlusions when the hand covers up UI elements [139, 140], and varying finger-to-screen ratios [39].

Many of these factors are spatially dependent (*e.g.,* the finger occludes more of the screen when it makes contact with the touchscreen at an extreme angle). Based on common hand postures used to grip a smartphone with one hand, Le et al. [177] characterized the region of the phone that could be comfortably reached. Mayer et al. [223] give further evidence through their analysis of 45 million touch events collected during touch-based gameplay, where they identified a region of the screen most likely to be comfortable to tap, known as the “sweet spot.” This suggests the utility of a spatially-dependent approach to personalizing touch interactions by modifying the position and size of UI elements.

2.2.2 UI Understanding

Datasets for UI Modeling

Several datasets have been collected for the purposes of analyzing and modeling mobile UIs. The Rico dataset is a large dataset of 72,000 mobile UIs and associated metadata including view hierarchies, screenshots, and user interactions, collected from 9,700 publicly available Android apps [81]. The FrontMatter dataset uses static analysis techniques to predict the purpose of UI elements by determining which system APIs are invoked [171]. Large datasets like these have enabled ML-based methods which can perform various tasks involving mobile UIs, including providing accessibility annotations [197, 325], giving design feedback [146, 278, 295, 341], suggesting common interaction flows [351], summarizing screens [315], automating interaction with UIs [29, 196, 280], and creating rich embeddings of UI image and text data for other downstream tasks [32, 135, 193]. Almost all currently available datasets are manually created in some aspect – through manual user interactions with UIs, and/or human annotations. This suggests the need for a more sustainable and potentially automated method of collecting UI examples and labels. This would allow ensure that datasets used to train UI understanding models is kept current to keep up with evolving UI design trends.

Predicting Screen Semantics

Computational representation of user interfaces are useful for many downstream tasks, such as design assistance [181, 213], accessibility improvement [347], and task-oriented systems. Screen Recognition [347] generates accessibility metadata of a UI from screenshots using an object detection model and heuristics. Several models [62, 102, 225, 342] have also been trained to predict the semantics of unlabeled icons found in mobile apps. These models can be applied to improve the accessibility of mobile apps, either as a tool during design time or as an automated system that repairs existing apps at runtime. Most of these models map UI elements to a pre-defined set of classes (*e.g.* UI element and icon type), which may exclude less common components [64].

An alternative is to train models using self-supervision [80, 107, 192], which allows them to take advantage of larger unlabeled datasets. Screen2Vec [192] and other pixel-based autoencoders [80, 213] map UIs to fixed-length embedding vectors which can be used to represent semantic properties. The Pixel-words model [107] employs a transformer model architecture and masked training objective based on prior work in NLP [84].

Reverse Engineering UIs

Many approaches to visual UI modeling focus on “reverse engineering” hidden attributes and potentially modifying them at runtime. Reverse-engineering methods often focus on extracting semantic attributes from visible information presented by the app (*i.e.,* pixel information), which allows them to support a broader array of use-cases.

An important use-case is facilitating non-visual access to apps for people with disabilities. Outspoken [279] was of one the first screen readers that supported GUIs, which required it to describe both text and graphical elements of the screen. To process icons and other pictorial elements, the system maintained a database of graphical elements (paired with a verbal description) and matched on-screen elements to descriptions of similar items. Today, the ecosystem of UI toolkits is much larger and permits much greater functionality, including allowing developers to embed icon and image descriptions in an app's metadata, yet many apps are still inaccessible because they do not include this data. To support inaccessible apps, recent screen reader technology [62, 347] uses deep convolutional neural networks to generate element descriptions and other accessibility metadata.

Reverse engineering methods can also be used to extend existing GUI applications. A common approach to interface with applications without an application programming interface (API) is to define "macros" that automate sequences of key-strokes and mouse movements. To acquire interaction targets, many automation toolkits provide functions for searching the screen for pixel values and returning their coordinates [30]. Sikuli [340] and PAX [57] are systems that improve the localization of targets by supporting more advanced matching techniques (*e.g.,* bitmap matching and heuristics) and combining hierarchical information extracted from an external source, such as the system window manager. Elements localized using pixel-based methods can also be used to modify apps at runtime [86], and previous work has investigated the benefit of hierarchy prediction (using heuristics) for this use-case.

Finally, reverse-engineering approaches have been applied to generate code from UI mockups or screenshots. A subset of these approaches have focused on translating hand-drawn wireframes to GUI code. These tools [25, 174] are useful for designers who wish to quickly sketch and prototype possible UI layouts. A more complex version of this task is generating code from complete UI screenshots, as it requires that the system handle the stylistic and structural variation present in real-world app screens. REMAUI [238] is a system that uses heuristics to combine OCR detection results and cropped patches from the original screenshot to generate working UI code. Pix2Code [38] is an end-to-end code generation model that uses a CNN encoder to encode a screenshot and a RNN decoder to generate code. UI2Code uses a similar architecture to generate a "GUI Skeleton" from a screenshot [61] that describes the relative positioning of UI elements.

Defining and Extracting UI Models

While reverse-engineering systems can effectively predict a subset of a screen's attributes, automated systems aimed at deeper and more complex interactions with UIs must support higher-level, semantic understanding of UIs. We reviewed literature related to model-based user interface (MBUI) development, which here we use as a conceptual framework for describing how UIs are constructed, presented, and used.

MBUI development refers to a development process that *(i)* first defines high-level models for an interface, then *(ii)* produces code that conforms to that model [125]. Models, among other things, detail what data the UI will display and how it will be used, and are a helpful tool for organizing the creation of UI applications. Puerta [257] describes an example of how this process is applied and categorizes common types of models (*e.g.,* data model, domain model, presentation model). Because a well-designed model can describe all or nearly all aspects of an interface, it is often possible to automatically generate code from model specifications [110, 239].

Similarly, it may be useful for automated systems to extract or infer models from a finished application, as doing so would reveal semantics. This suggests the need for systems or models capable of visually inferring structured relationships, which could be translated to UI models.

Structured Prediction from Visual Information

Many approaches to structured prediction have their roots in natural language processing (NLP). Early work on scene segmentation used stochastic grammars to analyze layouts (known as geometric parsing [283]) or construct hierarchical representations from proposal regions (*i.e.,* image patches) [310, 353]. However, it can be difficult to define or induce a grammar that explicitly describes all primitives and relationships and work well with continuous attributes. Moreover, many grammars are designed to work with sequential input (common in language) rather than spatial input (common in vision). Socher et al. propose a more general architecture that learns to recursively join related items in both images and text using a neural network model [290].

More recent work in the computer vision literature has focused on visual scene understanding through *scene graphs*. Scene graphs represent relationships between objects detected in an image and are described as a collection of relationship triplets (¡subject, predicate, object¿) [162]. Approaches to scene graph detection vary – some models first perform object detection then consider all possible pairs [343], while others directly generate a set of likely relationships [338].

Predicting UI Interactions

An important application of large annotated UI datasets is supervised training of machine learnings that predict UI semantics. For example, in this paper we focus on the problems of element tappability [278, 295] and screen "fingerprinting" [101]. More recently, Reinforcement Learning (RL) has been applied to model user interactions with both physical and digital interfaces. Oulasvirta et al. proposed a general framework based on RL of how users incorporate cognitive facilities, their experiences, and their environment in understanding and interacting with computers [248]. Under this context, an important part of knowing how to interact with an interface is by understanding its affordances. Affordances are the functional properties of an object (e.g., UI) that suggest how it should be used [244], and designer commentary suggests that design patterns can make affordance discovery more difficult. Liao et al. used a virtual robot agent equipped with sensors to simulate and learn how humans may discover affordances in physical interfaces (e.g., buttons and sliders) [205]. Our work aims to achieve similar goals of learning the affordances (e.g., tappability) and capabilities of interfaces.

Continual Machine Learning

A promising direction for training UI understanding models is to continually learn about UIs over time through sustained, potentially endless interaction. This approach is related to active learning (specifically online active learning), which is a field of ML that seeks to improve models using only a limited number of human-labeled examples [122]. These approaches often identify and prioritize difficult or representative examples to produce the best possible model from a small dataset. Another related approach is Never-Ending Learning, which is an ML paradigm for creating systems that continually learn from acquired experience rather than a single dataset. It was first applied to web-based knowledge using the NELL system [228]. The system has been running for prolonged periods of time (years) and has accumulated over 50 million beliefs (i.e., hypothesized knowledge snippets), which is possible only by processing large amounts of data that are prohibitively expensive to annotate. This learning approach introduces unique challenges, such as the need to learn from new data while retaining previously acquired knowledge. There are several techniques in the literature that can be applied to retain previous knowledge that involved i) regularization [161, 202], ii) rehearsal-based approaches [263], and iii) techniques that address task-recency bias [56]. From a practical standpoint, implementation also necessitates maintaining large ever-growing datasets collected over time, which could either be addressed through a robust crawling infrastructure or using dataset distillation methods that keep the most relevant samples [236, 237, 318].

2.2.3 UI Generation

Prior Approaches to UI Generation

Many computational tools and approaches have been developed to support the UI design and authoring process. One approach is to use optimization-based approaches to generate a UI that maximizes an objective function. Sketchplore [306] and Scout [297] were UI prototyping/design tools that integrated a layout optimizer to generate design suggestions. Simlar approaches have been integrated earlier in the design process (e.g., to produce diverse starting points) [77], and have even been used to refine existing designs [90, 327]. Neural networks have also been used to complete partially complete layouts [195] and generate layouts “from scratch” [187] or other conditional input [67, 188].

Instead of generating UI layouts directly, another approach is to first generate code and then use a UI toolkit to render the resulting UI. Model-based UI (MBUI) development is an example of an approach that converts high-level specifications of a UI’s properties or data into lower-level code. MBUI has been successfully applied in many applications, especially for automated interface generation [258], simplifying the implementation of complex adaptive UIs [239], or personalizing UIs [319]. Instead of generating UI code from abstract specifications, other approaches used existing sketches or UI mockups as input. SILK was an early system that used computer vision methods to detect sketched elements and translate them to code implementations [174]. Recent approaches have built on this approach by applying more sophisticated methods of inferring UI layout and hierarchy from visual input [38, 61, 180, 326].

Systems that rely on visual input may still introduce a usage barrier because it requires a screenshot or UI mockup as input. An alternative is language-based code generation, which can be more appropriate for early-stage design exploration or novice use. Several systems have been developed to retrieve relevant UI exemplars [50] or code [270] from databases using natural language requirements [164], descriptions [147], or conversation [276, 307].

UI Quality Evaluation

Traditional UI evaluation, initially rooted in heuristic evaluation and established guidelines [153, 243], has evolved significantly over time. The development of automated metrics marked a transition towards more objective and scalable UI assessments. Early work like ARNAULD [111] collected *user preferences* about specific outcomes to automatically learn and tailor a cost function for UI assessment and adaptation. tLight [227] continued this vision and presented eight automatic metrics for evaluating graphical user interfaces' aesthetics, demonstrating their effectiveness on desktop and mobile platforms. Progressing further, researchers also explored assessing the visual complexity of mobile user interfaces, establishing metrics that link visual complexity to perceived usability [269]. This shift underscores a growing emphasis on quantifying user interface elements to predict usability outcomes. Moreover, integrating cognitive principles into UI evaluation is gaining traction, with metrics now considering harmony and attractiveness, aligning with how users perceive and organize visual information. For instance, the Aalto Interface Metrics (AIM) service [246] demonstrates how blending user perception and attention models can improve GUI design evaluation. In recent developments, deep learning has been employed to model user interaction aspects like tappability [277, 294] and draggability [328], marking a shift towards using neural modeling to enhance our understanding of user behaviors. Furthermore, with the rise of generative models, recent research applies Large Language Models (LLMs) to provide UI design feedback [91], illustrating how combining design knowledge parameterized in large pre-trained models with user input can be helpful for designers to improve the visual UI design.

Machine Learning-based Quality Metrics

Learning scoring functions has been an important topic in many areas of machine learning. In the context of text-generation or machine translation, a popular class of text quality metrics involve the use of "ground truth" responses known as "references." BLEU [251] and later variants like ROGUE [209] and Meteor [35] were developed for other applications, such as summarization [118]. However, not all domains have access to human-authored references, leading to the development of "reference-free" metrics. Perplexity [154], for instance, is a classic metric used to estimate how likely a piece of text, often generated by a machine, is to come from a human-generated corpus. More recently, direct human evaluations have been utilized to assess model-generated text [349]. This type of evaluation system often ask individuals to compare outputs from the same input text to determine which model-generated version they prefer. In the realm of computer vision, numerous metrics have been devised to evaluate the quality of images produced by models, including the inception score [271], the Fréchet Inception Distance (FID) [137], and more recently the HyPE scores [350]. Finally, there has been a class of evaluation methods aimed at multi-modal applications that concern both text and images. CLIP-

Score [136] is a technique for assessing the quality of image captions by using the pre-trained OpenAI CLIP model. CLIP-IQA [317] further adopts CLIP to contrastively learn a function that evaluates images based on various quality attributes (e.g., brightness, colorfulness) and perceptual aspects (happy, scary). TIFA [144] introduces a method where it asks and answers its own visual questions using large vision-language models. It then quantifies how well the text prompts and the images generated from those prompts align. However, current, off-the-shelf vision-language models like CLIP often fall short in accurately analyzing UI screenshots, particularly when assessing UI design quality through language, due to the poor representation of UI examples in their training datasets.

3 Recommending Accessibility

3.1 Preamble

Content in this chapter is based on a conference paper that was published at Web4All 2021 [324].

While previous research has focused on building completely adaptive UIs (*e.g.,* SUPPLE [110]), that generate UIs from high-level specifications or objectives, in this chapter, I frame UI adaptation as an interface configuration problem. Feature-based adaptation has several benefits. Most smartphone OS's have dozens of built-in accessibility features (50+ on iOS), so there is already a very large of potential UI variations that can be reached by feature combinations. While many features result in seemingly simple changes (*e.g.,* changing global typeface), they are often designed by accessibility experts to be effective for the targeted user groups. Some features can result in substantial changes to how apps are used; for example, Voice Control is a feature that allows graphical apps to be used completely through voice. Yet, feature-based adaptation also has drawbacks. Because many third-party apps are developed using inaccessible third-party toolkits, they do not properly expose application semantics that allow them to respond to feature settings or assistive technology. Later chapters in this dissertation will focus on repairing these missing semantics through computational modeling. Another drawback is that users need to know about them in order to properly configure them. In this chapter, I present a system that recommends existing accessibility features included in popular smartphone OSes to users based on observed usage behaviors.

Numerous accessibility features have been developed and included in consumer operating systems to provide people with a variety of disabilities additional ways to access computing devices. Unfortunately, many users, especially older adults who are more likely to experience ability changes, are not aware of these features or do not know which combination to use. In this paper, we first quantify this problem via a survey with 100 participants, demonstrating that very few people are aware of built-in accessibility features on their phones. These observations led us to investigate accessibility recommendation as a way to increase awareness and adoption. We developed four prototype recommenders that span different accessibility categories, which we used to collect insights from 20 older adults. Our work demonstrates the need to increase awareness of existing accessibility features on mobile devices, and shows that automated recommendation could help people find beneficial accessibility features.

3.2 Introduction

Over the past decades, numerous accessibility features have been developed for people with a wide variety of abilities to use computing devices. Screen readers present otherwise visual content audibly, zoom features enable people to see content better, switch controls allow people to navigate screen content with switches (triggers), features tune out noise and allow users to hear more of what matters, and reading support adds an auditory component to text while reading and/or writing. Although many features exist, it is unclear how or whether people find the accessibility features that they could benefit from.

There is reason to believe that many people do not know about these features, and thus may not discover these features when they need them. As an example, one study found that only 1 of 50 participants were aware of the zoom feature in their web browser [45]. Another study found that only 3 of 14 older adults were aware that many mobile devices now come with accessibility features [106]. Older adults present a case worth further investigating, as many of them may not consider themselves as having a disability, but may nevertheless benefit from accessibility features as they age [9].

In this paper, we explore methods for matching accessibility features for people who may not know they could benefit from them. A natural starting point is to consider whether recommender systems may help with discovering accessibility features. One challenge is obtaining the data needed to construct these models in a privacy-preserving way. Health-related, and especially disability-related, information is highly sensitive, and users may be unwilling to provide this data [10]. Moreover, common recommender systems work via collaborative filtering [274], which leverages the idea that users who like certain things will like similar things. This could work in the area of accessibility recommendation, *e.g.*, if a user has turned on the VoiceOver screen reader, we might infer that they might also want to turn on Audio Descriptions because other users often have that pair of features turned on together. Yet, one of the assumptions in this project is that many people will not know to turn on accessibility features at all. If a person has never turned on any accessibility feature, there is no usage data to even start the first recommendation (*i.e.*, the “cold start” problem in recommender system research [173]).

An alternative approach that we advance in this paper is to recommend accessibility features based on how a user is interacting with a device. For instance, if the user is holding the device closer (or farther) than we would expect, that might indicate that they are having trouble seeing it, and could thus benefit from a font size increase. Likewise, if users are unable to perform double-clicks fast enough for the gesture to be recognized, then we might suggest to them the feature that allows more time between clicks. This is not an entirely new idea, *e.g.*, if one presses the shift key on Microsoft Windows repeatedly (perhaps indicating they are having difficulty using it as a modifier key), then Windows will ask the user if they would like to turn on “Sticky Keys”, an accessibility feature introduced in Apple System 6 (1988) that turns the modifier keys into toggles so key combinations can be pressed one key at a time [17]. It is also possible to detect stuttering in speech, which may eventually be connected to features to make speech recognition work better [179]. As we will present, many of the accessibility features available on today's smartphones can be activated using such mechanisms derived from behaviors detected based on device use.

The main assumption behind our approach is that users should use their device as they normally would in order to understand their usage patterns and receive recommendations for accessibility features. This differentiates our work from past work on, *e.g.*, digital games specifically designed to detect dyslexia [266, 267] or autism [313], and wizards that explicitly ask users to describe their accessibility needs. Therefore, our approach may not be able to recommend some accessibility features to a subset of users. For instance, our approach may not be able to recommend VoiceOver to a blind person, who has never heard of the VoiceOver screen reader feature. However, we might be able to recommend VoiceOver to a user who is using the zoom feature at a high zoom level and still holding the device closer than would be expected. Most accessibility features do not dramatically change smartphone functionalities and are amenable to our approach of recommending based on observed usage.

In this paper, we first present the results of a survey with 100 participants (including 25 people over the age of 50) demonstrating that very few people are aware of available accessibility features on their devices. We then select four common accessibility features, each selected from a main category of accessibility features (*e.g.,* vision, hearing, interaction and mobility), and develop prototype recommenders for them. We initialized our recommenders from a baseline study with 10 participants, and then used them to explore accessibility recommendation with 20 participants.

Our paper makes the following contributions:

- We demonstrate and quantify that awareness and knowledge of how to use accessibility features is low among smartphone users (one-fifth of users knew what "accessibility" means) and even lower among adults over the age of 50 (one-tenth), who are more likely to benefit from them.
- We show that many existing accessibility features are conducive to recommendation, and we provide recommendation strategies for detecting relevant usage behaviors. Using these strategies, we constructed four prototype recommenders spanning accessibility categories.
- We conducted a study with 20 participants to collect insights on the utility and preferences of accessibility feature recommendation.

3.3 Accessibility Awareness Survey

We conducted an online survey to understand people's awareness of accessibility features on their mobile phones. Specifically, we were interested in how people might react if they developed an accessibility need, and if they would know to use (or even check) their phones for features that could help. A challenge in investigating awareness of accessibility features is that questions directly asking about a feature, *e.g.*, "Did you know you can make the font size bigger on your phone?", may make respondents aware of the feature. To address this challenge, we used a staged reveal approach in which we first asked about hypothetical situations a feature could be useful in to see if participants mentioned accessibility features as a potential solution, and then later asked how they would go about configuring accessibility features to better support their needs.

In our survey, we asked participants about their smartphone usage and a series of questions aimed at assessing their awareness of accessibility features. We obtained 100 survey responses and collected demographic information (42M/58F, ages 19-83, mean age 42.4). Our sample included 25 adults above the age of 50, who are significantly more likely to develop a disability in the near term [9] and could benefit from accessibility features in the future. The survey was conducted through an online polling platform Pollfish [8] and targeted a general population in the United States. We asked respondents what type of smartphone they used most frequently — 42% used Apple iOS while 58% used Google Android. Most respondents (74%) were frequent smartphone users (*i.e.,* used their smartphone multiple times an hour), and 95% of respondents reported using their smartphones at least a couple of times per day. Furthermore, participants reported using a wide range of apps on their smartphone, with respondents reporting using Lifestyle (43%), Social Media (76%), Education (18%), Games/Entertainment (54%), Productivity (35%), Utility (56%), and News/Information (51%) apps. 62% of respondents reported wearing prescription glasses or contacts, and 18% reported that they were sometimes unable to hear clearly without the use of a hearing aid.

Our survey investigated the following research questions:

RQ1 - Would users think to check their mobile device if they developed an accessibility need?

RQ2 - Do users know how to configure their devices to better support accessibility needs?

To address these research questions, we included two types of questions in our survey: *(i)* hypothetical questions (**RQ1**), and *(ii)* feature-based questions (**RQ2**). Figure 3.1 shows the specific hypothetical scenarios and features used in these questions, which were chosen to span features from various categories (e.g., vision, hearing, etc.) and be plausible candidates for recommendation.

To answer **RQ1**, we first asked our participants how they would use their phone in hypothetical situations where they encountered certain types of impairments related to accessibility features. For example, we asked participants: "*Imagine your eyesight gets worse so you can't easily read what's on the phone screen, what would you do?*". This allowed us to infer awareness of these features, without revealing their existence in the question itself. In total, our survey contained 7 of these hypothetical questions (Figure 3.1), each corresponding to a different accessibility feature. Following the hypothetical questions, we presented feature-based questions (**RQ2**) that provided high-level solutions to some of the previously posed hypothetical situations but required respondents to demonstrate knowledge of feature usage. For example, we asked "*How can you make the content on the screen larger and easier to view?*". As a part of this set of questions, we also asked participants to describe what the function of the Accessibility menu was in their device's settings. In total, our survey contained 5 of these feature-based questions (Figure 3.1). Survey-takers were told to answer these questions using their existing knowledge of smartphone features (or indicate "I don't know") and were explicitly told not to search for answers online or check external resources. In summary, we purposefully staged our questions to gauge accessibility awareness without initially "giving away" the existence of accessibility features (*i.e.,* hypothetical questions); then, we "narrowed in" on specifically asking for device-based solutions (*i.e.,* feature-based questions).

Responses for all questions were coded by 3 researchers trained in HCI and qualitative methods. The responses were coded using the following categories.

- *Correct setting* (**C1**) - The response provided the correct accessibility setting (*e.g.,* Enable Larger Text in the Accessibility menu) or another feature that provided the same level of access to device content (*e.g.,* Change the system font size in the Display Settings). Solutions for any mobile operating system (*e.g.,* iOS, Android) were marked correct.
- *Other Smartphone-based solutions* (**C2**) - The response provided a solution on the smartphone but was either too vague (*e.g.*, Change the settings) or did not work in all cases the accessibility feature did (*e.g.,* Double-tap to zoom in). Responses in this category often indicated that the user was aware that their device was capable of making content more accessible, but did not know how to enable that functionality without additional help. One of the problems is that users need to know the name of the feature beforehand in order to search for it, and our goal is to proactively surface them to the user to remove this need.
- *Other* (**C3**) - The response provided did not demonstrate any knowledge of awareness or usage of mobile accessibility features, but, as we discuss later, can still constitute a valid course of action.

Hypothetical Questions (**RQ1**) - To analyze responses from our set of hypothetical questions, we wanted to see how many participants' responses mentioned using functionality already present on their smartphone (**C1**, **C2**). We coded the responses to the hypothetical questions with an inter-rater reliability (Fleiss' Kappa) of $\kappa = 0.68$. While there was some disagreement (between the **C1** and **C2** codes) on whether a response provided "the same level of access to device content" as the correct accessibility setting (*e.g.,* some responses mentioned using voice commands for questions targeted at touch accommodations), it was clear that users did not think to check their mobile devices for accessibility affordances (**RQ1**). Only 15.7% of participants would have attempted to look on their phone for a solution and only 12.1% identified the most effective setting (Figure 3.1).

We also more closely analyzed the **C3** category to better understand those responses. Unsurprisingly, a common response was that participants stated they would consult with a doctor or medical professional. For example, if their eyesight prevented them from easily reading on-screen content, they responded that they would go to an optometrist to get glasses. As previously mentioned, human experts can match access technology with a higher degree of certainty and effectiveness. However, for many reasons (*e.g.,* time/money requirements, doctor's lack of knowledge of a patient's device, non-medicalized accessibility need such as slightly degraded vision) this method may not lead to the widest adoption of accessibility features. Other participants proposed purchasing additional software or equipment to access content (*e.g.,* buying a loud Bluetooth speaker when unable to clearly hear content or buying a new phone). Finally, another common type of response included asking someone else to help perform the action or trying again slowly, but these would not be feasible in many situations.

Feature-based Questions (**RQ2**) - We performed a similar coding for our feature-based questions ($\kappa = 0.73$), but because participants were required to consider their smartphone as a part

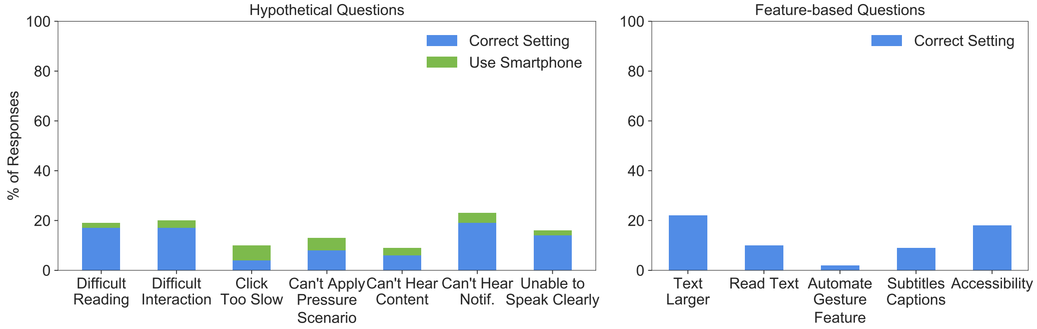


Figure 3.1: This figure shows a summary of responses to our online survey, which was composed of hypothetical (Left) and feature-based (Right) questions. For each of set of questions, we coded the responses to show people's knowledge of smartphone accessibility features. From our set of hypothetical questions (Left), we find that on average, 15.7% of participants would have attempted to find a solution on their smartphone and only 12.1% identified a setting that addressed the scenario. When asked which settings/features were needed to make certain content more accessible (Right), participants responded correctly 10.3% of the time, and only 18% of participants knew what "accessibility" meant in the context of their phone's settings.

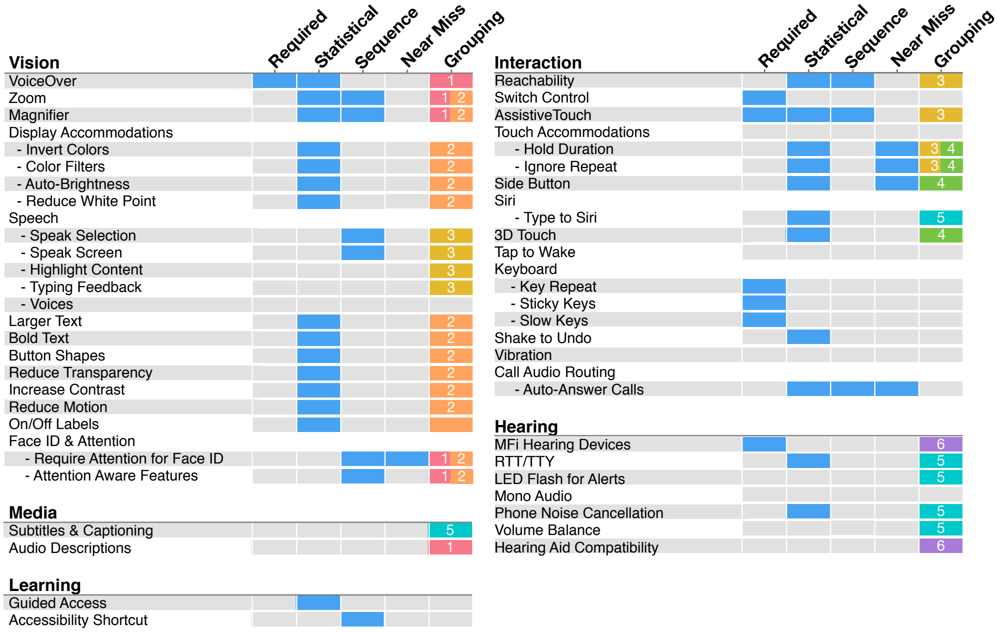
of their solution, we only focused on whether the response provided the Correct setting (**C1**). The responses from the feature-based questions indicate a similar conclusion, showing that on average only 10.3% of responses by participants mentioned the correct setting (or an alternative solution providing equivalent utility). Finally, to more directly answer **RQ2**, we asked participants to respond with the definition of "accessibility" in the context of their phone's settings, and only 18% responded correctly, suggesting that most people would not know how to access their devices' built-in accessibility capabilities.

In summary, the results from our awareness survey show that although users rely on their smartphones for a wide variety of tasks, they are generally unaware of the accessibility features available on their smartphone. While one might suggest that users seek out this information when they develop the need, they may not know to look, and continue to "get by" using their device (*e.g.,* squinting or bringing the phone closer to their eyes). Thus, we believe a smartphone that proactively recommends accessibility features would improve users' interactions with their devices.

3.4 Recommending Accessibility

To chart the design space of how accessibility features could be recommended, we reviewed and categorized a set of features available on modern mobile platforms. For our exploration and building proof-of-concept prototypes, we scoped our effort to iOS 12 [1], which contains nearly 50 accessibility features (Figure 3.2). Many of its accessibility features (*e.g.,* font size adjustment, content magnification) are standard across other platforms, so our recommenders can be directly transferred. Our approaches to building recommenders (*i.e.,* strategies for feature

[1] https://www.apple.com/accessibility/iphone/



Vision	Required	Statistical	Sequence	Near Miss	Grouping
VoiceOver	🟦	🟦			1
Zoom		🟦	🟦		1 2
Magnifier		🟦	🟦		1 2
Display Accommodations					
- Invert Colors		🟦			2
- Color Filters		🟦			2
- Auto-Brightness		🟦			2
- Reduce White Point		🟦			2
Speech					
- Speak Selection			🟦		3
- Speak Screen			🟦		3
- Highlight Content					3
- Typing Feedback					3
- Voices					
Larger Text		🟦			2
Bold Text		🟦			2
Button Shapes		🟦			2
Reduce Transparency		🟦			2
Increase Contrast		🟦			2
Reduce Motion		🟦			2
On/Off Labels		🟦			🟧
Face ID & Attention					
- Require Attention for Face ID			🟦	🟦	1 2
- Attention Aware Features			🟦		1 2

Media	Required	Statistical	Sequence	Near Miss	Grouping
Subtitles & Captioning					5
Audio Descriptions					1

Learning	Required	Statistical	Sequence	Near Miss	Grouping
Guided Access		🟦			
Accessibility Shortcut			🟦		

Interaction	Required	Statistical	Sequence	Near Miss	Grouping
Reachability		🟦	🟦		3
Switch Control	🟦				
AssistiveTouch	🟦	🟦	🟦		3
Touch Accommodations					
- Hold Duration		🟦		🟦	3 4
- Ignore Repeat		🟦		🟦	3 4
Side Button		🟦		🟦	4
Siri					
- Type to Siri		🟦			5
3D Touch		🟦			4
Tap to Wake					
Keyboard					
- Key Repeat	🟦				
- Sticky Keys	🟦				
- Slow Keys	🟦				
Shake to Undo		🟦			
Vibration					
Call Audio Routing					
- Auto-Answer Calls		🟦	🟦	🟦	

Hearing	Required	Statistical	Sequence	Near Miss	Grouping
MFi Hearing Devices	🟦				6
RTT/TTY		🟦			5
LED Flash for Alerts					5
Mono Audio					
Phone Noise Cancellation		🟦			5
Volume Balance					5
Hearing Aid Compatibility					6

Figure 3.2: 48 accessibility features listed in the iOS Accessibility menu, labeled according to our detection strategies: *Required* are features that require hardware or whose users would be unlikely to use the device without them; *Statistical*, *Sequence*, *Near Miss*, and *Grouping* refers to the detection strategies. For every feature (row), a strategy (column) is highlighted if it is applicable. For features using the *Grouping* strategy, we also indicate which group they belong to (using a group number and color). For example, if the user has VoiceOver enabled, other features in that group (*e.g.,* Audio Descriptions, Require Attention for Face ID) can be recommended.

recommendation) can also be applied more generally to features that we did not initially explore.

A few of these features either require special hardware (*e.g.,* hearing aids) to be connected in order to be used, or are meant to address accessibility needs that would likely prevent users from using the device without them (*e.g.*, VoiceOver), and so we would not be able to recommend these features. This is not intended to be an exhaustive or complete list, necessarily, but is rather the result of iterative process among the authors (accessibility and sensing experts) to identify promising directions for each feature. Generally, our approach to recommending accessibility involves *(i)* identifying an accessibility need (potentially corresponding to an available feature), *(ii)* developing a hypothesis about how it might manifest in device signals, (*e.g.,* sensor readings, system events), *(iii)* determining a detection strategy to decide when to surface a recommendation to the user, and *(iv)* empirically validating, to the extent possible, that the detection method works as intended with acceptable accuracy. In our work, we specifically focus on detecting potential accessibility needs from observed usage data. While detection constitutes a large part of recommendation, there are other aspects (*e.g.,* how to surface, strategies for increasing adoption) that we leave to future work.

We categorize accessibility detection into four approaches: statistical, near-miss, action sequences, and grouped detections. In this section, we give a brief description of each method along with an example use-case.

Statistical

Statistical detection involves identifying differences in users' behavior statistically over time. This approach is useful when one or more signals are known to be relevant, but it is unclear what specific bounds or behaviors to detect. For instance, users may not realize that they frequently hold the phone close to their face to read content on it or that they consistently listen to media at a high volume. This approach generally leverages statistical tests and outlier detection algorithms to compare an individual's usage patterns with a pre-defined range. This detection method draws from prior approaches, such as machine learning techniques for dyslexia detection [266] and ability detection [109, 113, 150]. Generally, such approaches have assumed labeled data for supervised machine learning or optimization algorithms; yet, we find that even when using simpler approaches with fewer data (*i.e.,* univariate statistical tests), we can successfully detect relevant behaviors.

> *Font Size Increase* – if a user tends to hold the phone closer (or farther) from their face than the common distance we expect, then they may benefit from a larger font size.

Near-Miss

Another type of detection strategy is to monitor features that rely on a pre-defined threshold or require multiple conditions to be reached before triggering. Often, the default threshold values may be difficult to reach for people with disabilities (*e.g.,* the default speed for double-clicking the home button may be too fast for people with motor impairments), so surfacing accessibility options that allow the adjustment of these values can significantly improve the user experience. Monitoring the threshold values for features that have them and logging "near-misses" can be used to trigger recommendations. Often, the accessibility feature itself gives a good clue as to what to look for, *i.e.*, if a double-click needs to happen with no more than 1 second latency between button presses, looking for examples of two button presses in sequence with a slightly longer gap between may be a good signal that the user could use more time.

> *Side Button Click Speed* – double and triple clicking the side button on iPhone invoke Apple Wallet and a shortcut to other accessibility features, respectively. Observing a user press the button two or three times, and fail to activate the feature (when they would have succeeded if the speed had been set to slower), may indicate that they could benefit from setting the required speed to slower.

Action Sequences

In some instances, a strong connection is known between a specific sequence of behaviors and a feature that might be useful. Action sequence detection is implemented by monitoring and recording system events (*e.g.,* app open events and UI interaction events). This method is informed by prior shortcut induction from user behavior [75], and programming by demonstration systems that have been used for accessibility purposes [47, 48, 129]. In most cases users may discover that some task is inaccessible and perform action sequences as "work-arounds" to achieve the same functionality, sub-optimally.

Magnifier – the Magnifier in iOS allows users to use their phone's camera as a magnifier. We have informally observed people use an alternative way to access such functionality, which involves taking a picture of something (*e.g.*, a restaurant menu), opening the photo in the photo viewer, and then using "pinch to zoom". This sequence of actions could indicate that the user could benefit from the Magnification feature [14].

Grouped

Finally, grouped detection can be used to recommend new accessibility features based on the ones the user already has enabled. This is useful because related features do not always appear close to each other in the Settings menu, and some accessibility needs may not directly manifest themselves in signals detectable by other approaches. Grouping is a manual approach to replace recommendation algorithms, which do not have the necessary data to provide recommendations. Instead, we used simple conditional statements to recommend other grouped features. Unlike larger recommendation systems, the number of accessibility features to recommend is relatively small. A manually curated approach is manageable, although we envision grouped recommendations could potentially be data-driven if such data was available.

Type to Siri – this feature allows users to type their queries to Siri. This may benefit deaf or hard-of-hearing users, and could be recommended when users turn on Hearing-related features.

Other examples are much more straightforward because they are already grouped together, *i.e.*, if you use one of the Vision-related features you might also benefit from other Vision-related features. For instance, neither Audio Descriptions nor Type to Siri is grouped with Vision or Hearing, respectively. In Section 3.5.4, we provide more details on the groupings implemented by our prototype system. Figure 3.2 shows more potential groupings between accessibility features, uniquely grouped by an identifying number and color.

3.5 Prototype Recommenders

To move towards concrete implementations of recommenders, we used several recommendation strategies and applied them to some of the features discussed in the accessibility awareness survey (Figure 3.1). We first performed a baseline data collection with an initial group of participants using a popular consumer smartphone (iPhone XS) to understand how different usage behaviors manifested themselves in sensor data. We describe the procedure for this data collection in the user studies section. Then, using our recommendation strategies, we built four accessibility feature recommendation prototypes. Each prototype targeted one or more accessibility features that could be detected using similar strategies. These prototypes are exemplars, which demonstrate how we might go about developing future recommenders.

3.5.1 Font Size Recommender

Our font-size recommender prototype automatically senses when users find it difficult to read content and recommend features that would adjust the content's size. As noted by previous research on text magnification, adjusting font size can enhance the readability and experience for many users, especially aging adults [45].

We calculated viewing distance using the front-facing camera and ARFaceAnchor objects returned by the ARFaceTrackingConfiguration [3] and used the *Statistical* detection strategy to surface recommendations to the user if they were found to hold the phone outside of an expected viewing distance range. We chose to define our expected viewing distance range empirically, based on our baseline data collection ($M_d = 0.36m, \sigma_d = 0.049m$). Our results align with previous work quantifying average font size and viewing distance for smartphone content [31]. We triggered a notification recommendation when the difference between the user's mean viewing distance and M_d exceeded a threshold, which we conservatively set to two standard deviations.

3.5.2 Subtitles & Captions Recommender

Our "Subtitles and Captions Recommender" monitors device volume levels to recommend hearing accessibility features, similar to other features such as watchOS decibel meter, which does so for environmental noise [5]. We implemented a background daemon that continuously monitored 1) whether audio was currently playing, 2) the volume level, and 3) the output device. In the data collected from our baseline study, the average volume level was $M_v = 47.1\%, \sigma_v = 16.3\%$. Using the *Statistical* recommendation strategy, we surfaced a recommendation for the Subtitles & Captions feature when the user's listening volume was statistically greater (by a minimum of two standard deviations) than our baseline mean.

3.5.3 Side Button Click Speed Recommender

While touch interaction is the primary mode of interaction for most mobile devices, several important features require the use of physical buttons. These include adjusting output volume, locking/unlocking the device, and certain application-specific uses (*e.g.,* confirming an app installation) [16].

The default double-click speed on the side button can be difficult to trigger for many users with even slight motor impairments. Recognizing this, the time allowed between clicks can be changed (increased) via the accessibility menu [15]. Our prototype recommends this feature to users when it observes a "near-miss" failed attempt. To do this, the recommender monitors repeated button presses that occurred within the slowest possible double-click threshold. The recommendation is made if the input is too slow to trigger based on the current threshold, but would have done so using a slower setting.

3.5.4 Grouped Recommenders

Usage-based recommenders may be able to educate users about the existence of accessibility features, and then *Grouped* recommenders could help them expand and/or customize selected accessibility settings. Figure 3.2 shows a comprehensive grouping of accessibility features, while our grouped recommender prototype implements a subset of these using the iOS UIAccessibility API [12].

- AssistiveTouch → Side Button — If AssistiveTouch is enabled, the user might also benefit from the Side Button setting which can also make wider range of interactions accessible.
- Closed Captioning → Type to Siri — A Closed Captioning user may wish to interact with Siri using an alternative text-based modality.
- Bold Text → Larger Text — Similar to Bold Text, Larger Text increases the readability of on-screen text content by increasing the available font sizes.

3.6 User Studies

We conducted two user studies: *(i)* a baseline data collection and *(ii)* a user study. As mentioned earlier, the baseline data collection with 10 participants aided in the design of our detection strategies and initializing our prototypes (*e.g.,* triggering thresholds). We then used our prototype recommenders with 20 participants to generate insight on the utility and preferences for accessibility feature recommendation. All user studies were conducted before the COVID-19 pandemic, so there was no additional risk for participants.

3.6.1 Baseline Data Collection

Table 3.1: This table describes the tasks participants performed during the data collection study.

T #	Name	Description
1	Video Questions	Watch a short 3-minute TED talk then fill out a quiz (24 questions) on the smartphone.
2	App Installation	Use the App Store to install 5 applications. After downloading and installing, take a screenshot of the main screen, then uninstall the app.
3	Internet Scavenger Hunt	Answer 9 trivia questions about GPS technology using provided external links and a search engine. Record answers in a note-taking app.
4	Siri Questions	Answer 7 questions using Siri and record answers in a note-taking app.
5	e-Reader Questions	Find 10 pieces of information in a book chapter and record answers in a note-taking app.

Procedure

We recruited 10 participants (7M/3F, ages 24-40, mean age 32) for our baseline data collection study. Eight of the participants wore glasses or contacts with corrective prescriptions, and no participants used hearing aids. After obtaining consent, participants were given an iPhone XS device that was preloaded with a background daemon that recorded a variety of signals.

At the start of every usage session, the researcher reset some of the device's settings (*e.g.,* output volume, system font size) to the lowest possible value. This was done to encourage the participant to set these values according to their own preference rather than using the ones chosen by the previous person, and the researcher informed participants that they were allowed to adjust these settings. To simulate everyday usage, we gave participants a pre-determined list of common smartphone tasks (Table 3.1) to complete during the study session. The order of the tasks was randomized for each participant. Participants were given 45 minutes to complete the list of tasks, and the data collection session was stopped if the tasks were finished early. Afterwards, participants were given a shortened version of the accessibility awareness survey that

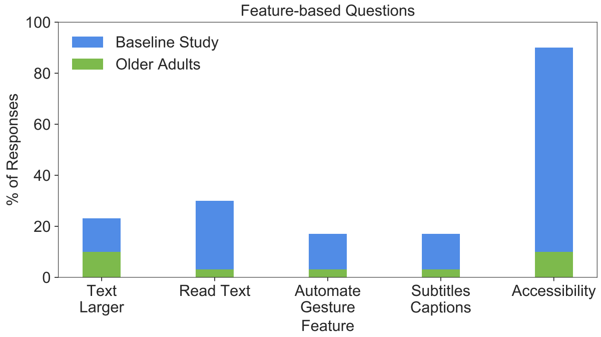


Figure 3.3: This figure shows the portion of responses from our in-person surveys (*i.e.,* Baseline Study and User Study) that provided the correct response to our feature-based questions. Awareness of features was much lower among adults over the age of 50, even though they were more likely to benefit from them.

included questions about demographics, mobile phone usage, and the feature-based questions. Participants were compensated with a $10 gift card.

The data obtained helped inform the design of our prototypes and provided a dataset to empirically set our recommender algorithm parameters.

3.6.2 User Study

To validate our approach and generate additional insights for accessibility recommendation, we conducted a study with two populations (n=20). All user studies were conducted before the COVID-19 pandemic, so there was no additional risk for participants. We were interested in answering the following research questions:

RQ1 - Were participants aware of the accessibility capabilities of their smartphones?

RQ2 - Did participants find the features recommended by our prototypes useful?

Participant Age Range

One of our motivations for this work was to surface recommendations for accessibility features to users who are likely to benefit from them (*e.g.,* older adults). In our recruitment of participants for this study, we settled on the age threshold of 50+, which we know is roughly the time when abilities start to really change [13, 314]. We acknowledge that this range is larger than most studies in HCI that study ageing, and likely encompasses several sub-groups (*e.g.,* older middle-aged, retirement-age adults, seniors) which have unique social norms, life experiences, and technology use. Our motivation is not bracket individuals into age groups or to create a solution specifically for one such group, but to identify broad segment of the population that can most benefit from accessibility recommendation.

Procedure

The first population (P1-P9) was recruited from a senior care residence and consisted of 9 participants (3M/6F, ages 79-97). We initially recruited 10 participants from our first population, but one later withdrew due to difficulty using the smartphone, and so we report on findings from the other 9 in that group. The second population (P10-P19) was recruited from a local participant pool with a minimum age requirement of 50 (5M/4F/1 Prefer not disclose, ages 50-79). Our participants had a diverse range of abilities, which allowed us to evaluate our system under circumstances experienced by a broad range of users. 78.9% of our participants wore glasses, and 36.8% participants used hearing aids. Most (78.9%) owned smartphones and reported varying frequency of usage — ranging from a couple of times per week or less (10.5%) to multiple times per hour (31.6%). When asked what kinds of apps participants used on their smartphones, some responded that they only used their phones for calling family members (5.3%), while others used Lifestyle (26.3%), Social Media (47.4%), Games/Entertainment (36.8%), Utility (57.9%), and News/Information (52.6%) apps. On average, participants reported that they had their smartphone for 4.3 years.

For both populations, we followed a procedure similar to the one used for our baseline data collection (Table 3.1). Users were asked to complete a set of tasks during a usage session, and then filled out a survey afterwards. In this study, we shortened the usage session from 45 minutes to 30 minutes by removing two of the tasks (Internet Scavenger Hunt and the e-Reader questions) to make time for a brief interview afterwards about participants' views on the recommendations. In addition, due to some participants' lack of experience using smartphones and motor impairments, we reduced the complexity of some of these tasks (*e.g.,* reducing number of questions on the video questionnaire). Although these procedural differences may impact the distribution of collected signals used for statistical detection (*e.g.,* viewing distance), we only removed tasks if they were similar to others in the set (*e.g.,* filling out an Internet Scavenger Hunt required typing text in the Notes app, as did the video questionnaire), and we did not observe a significant effect on our prototype's detection ability. For participants who preferred us to do so, we administered the post-study surveys verbally and recorded their answers for them in writing.

After administering the survey, researchers conducted a brief interview with participants structured around various accessibility features supported by our prototypes. We showed participants how to enable various accessibility features and demonstrated their effects on the user experience. We then asked participants to rate whether each feature was useful for them on a 7-point Likert scale (1: Strongly Disagree, 2: Disagree, 3: Somewhat Disagree, 4: Neutral, 5: Somewhat Agree, 6: Agree, 7: Strongly Agree). We concluded the interview by asking the participants if they thought they could benefit from accessibility features like these, and if so, how they would prefer the recommendations be surfaced. Participants were compensated $20 for their time.

3.6.3 Accessibility Awareness

Using the procedure for categorizing the questions from our online study, we coded the survey responses from both the Baseline Study and the Older Adults Study. The inter-rater agreement scores were $\kappa_{baseline} = 0.67$ and $\kappa_{senior} = 0.76$ for the responses from the Baseline Study and Older Adults Study, respectively. As the in-person surveys only included feature-based questions, we focused mainly on whether the responses fell into the Correct setting category (**C1**) or not. A comparison of the two populations can be seen in Figure 3.3. As seen in our analysis, most of our users were unaware of common accessibility features and how to use them. In answering **RQ1**, we found that there was very little awareness of accessibility features (or even what “accessibility” was) among older adults, even as they continue to engage with mobile technology (78.9% owned smartphones). Compared to the survey results from the baseline study, we found that awareness of accessibility features was much lower (10.5% of older adults knew what “accessibility features” were compared to 90% of baseline participants) among older adults, who more likely to benefit from them. Most of the participants in our Baseline Study were software engineers familiar with iOS and were more likely to know about accessibility features. Nevertheless, while 90% of participants generally knew about accessibility features, they were less familiar with specifics about how they could be used. From our Older Adults Study, we found relatively few participants knew about accessibility features (10.5%), even though they were more likely to benefit from them.

3.6.4 Utility of Accessibility Recommendation

Using the data collected from the participants’ usage sessions, we ran our detection strategies post-hoc (*i.e.,* participants did not interact with recommendations in real-time) to estimate the the utility of their recommendations. In total, our prototypes triggered 19 recommendations, and based on the participants’ ratings of the features, they would have found 73.7% of those recommendations useful, 21.1% not useful, and 5.3% neutral. This suggests that our participants would likely benefit from accessibility feature recommendations (**RQ2**). Our goal was to gauge participants’ perception feature recommendation and accessibility features overall, although additional work would need to be done to better understand the potential for these recommendations leading to adoption. Our conversations with participants support our initial analysis, and we aim to further strengthen this conclusion in future work.

Below, we further analyze each feature recommender in more detail and provide context for their performance. Because none of the participants had turned on any accessibility features, our Grouped recommender prototype is not applicable.

Font Size/Zoom

The Font Size/Zoom recommender was triggered by 3 of the participants (all 3 wore glasses), who, on average, gave those features a usefulness rating of 6.3/7. Interestingly, every participant in our study responded that they thought those features were useful when shown to them in our post-study interview. This may suggest that if such a feature recommender was deployed, it could be beneficial to decrease the triggering threshold or non-intrusively surface a recommendation for all users. Indeed, as part of the iOS new device setup process, a subset of display settings such as Display Zoom can be adjusted which affects the size of on-screen content. However, given that only two participants mentioned adjusting font size in their awareness survey responses, there is reason to believe that these features could be made more visible.

Subtitles & Captions

Similar to the Font Size recommender, the Subtitles & Captions recommender was also implemented by performing statistical detection on the user’s audio volume. The mean audio levels of 6 of the participants (4 required hearing aids to hear clearly) exceeded our threshold, and those participants rated the Subtitles/Captioning features usefulness 4.8/7 on average. While the majority of triggered recommendations were found to be useful, we found that observed signals did not always align with participants' ratings of features. For example, P4 watched the video at maximum volume but rated the Subtitles & Captions feature as providing low usefulness. When asked about the rating, P4 responded that he disliked watching TV and movies with closed captioning because he found them distracting. Other instances of declined recommendations may be explained by preference or by prior work on the attitudes of older adults toward disability and aging [149]. We took from this that an additional element to consider when recommending a new feature is not only the expected utility of the feature but also the user's acceptance of it.

Click Speed & Assistive Touch

Compared to the other recommenders, the Click Speed & AssistiveTouch prototype was triggered the most often, in part due to the more relaxed *Near-Miss* detection scheme used. While in practice, such a system might surface a recommendation after a couple of near-misses, we set our prototype to trigger after the first instance due to the short duration of the study. In total, 10 users performed a double-click at speeds which would not have been detected using the *Default* timing but would have using slower settings. Of these, 70% found the associated accessibility features useful and 30% did not. Among users who triggered the recommender, the average usefulness rating was 4.7/7. However, for users that triggered our prototype's *Slowest* threshold, all of them (100%) found the features useful. While *Near-Miss* detection is appealing in part due to its simplicity and direct connection to an adjustable setting, successful deployment of recommenders for features such as Click Speed require more robust schemes that may combine certain aspects of statistical detection (*e.g.,* modeling the number of near-misses for the average user) and take into account additional context (*e.g.,* it is the first time the user is performing the double-click gesture).

3.6.5 Additional Observations

Beyond Awareness

An interesting observation was that one participant (P19) knew about accessibility features, but described them as *"for the visually/aurally impaired"*, which suggests that because he didn't identify as having a visual or hearing disability, he would not have thought of looking for useful features under the accessibility menu. We believe that a proactive recommendation system such as ours could help surface features that provide utility to a broad range of users. Indeed, two of our recommenders were triggered by the participant's usage session, and both features were marked as useful in the post-study interview. The only other participant (P11) who correctly described what "accessibility" referred to responded *"Can't remember ... But [I've] used [them]... maybe to increase text/magnify"*, indicating knowledge of only a small subset of features that could potentially be useful. Among participants recruited at the senior care residence (P1-P9), we found that knowledge of accessibility features was non-existent (*i.e.,* none of the participants from this group knew what "accessibility" meant in the context of computing), even though many of them had begun to adopt mobile technology (77.8% of them owned a smartphone, and on average, they owned their smartphone for 2.3 years). Similarly, we believe that our system could provide a lot of value for these technology adopters by making certain tasks easier to learn or perform.

Recommendation Preferences

After conducting the study, the researchers were frequently asked by participants to show them how to enable certain features on their personal devices. Even for participants who did not trigger any recommendations (P18), when shown certain features (*e.g.,* Font Size), they indicated that they thought the features would be beneficial: *"I have perfect eyesight but I still have a pair of readers that I use sometimes [to reduce strain] ... a bigger font size would also make things easier to read."* Almost all participants (89.5%) were open to receiving recommendations, with most preferring low frequency surfacing methods (*e.g.,* home screen, email, or a message) that did not interrupt their current task. On the other hand, one participant (P2) indicated that she valued the potential usefulness of features over the interruption cost: *"If there is something that could help me use [my device], I want to know about it."* Furthermore, not all participants wanted these features to be recommended to them (P6, P8). While P6 agreed that accessibility features were useful for interacting with her smartphone, she preferred not to have them automatically recommended to her, saying *"I might find it confusing"*. P8 offered another reason: "*Not necessarily... Once I learn [how to do something], I'll be set in my ways*", stating that the novelty of interacting with the device through the recommended features might be off-putting. Similar to what we saw with declined recommendations, additional context such as user preference play an important role in user acceptance of accessibility features and new technology in general.

3.7 Discussion and Future Work

In this paper, we have introduced the idea of "recommending accessibility" as a fruitful area for research. Even as numerous useful accessibility features have started to be included in the smartphones that people own, our survey demonstrated that very few people know about them or know which of those features they could benefit from using. While on-device recommendation approaches are not the only useful strategy for building awareness, we believe that they are likely to be an important and necessary complement to existing advocacy and awareness approaches, especially as our devices, thankfully, contain more and more features intended to make them more accessible.

We have laid out a roadmap for recommending accessibility, explicitly outlining how accessibility features on iOS could map to a set of detection strategies and available signals that we believe largely characterize the space (Figure 3.2). We believe there is ample opportunity for future research in validating this across the many accessibility features on iOS and other platforms. Delivering a new recommender for a new feature will require substantial effort to validate, for instance, ensuring that the recommender works as intended and has an acceptable false positive rate in the real world. Ultimately, the success of a recommender approach seems dependent on a user deciding not only to turn on a feature but to adopt it; understanding adoption takes significant effort over time, especially given that each individual accessibility feature may only be expected to be useful to a relatively small percentage of people.

The prototypes introduced in this paper were designed as proof-of-concept implementations to demonstrate the potential of this direction and for use in our study with older adults. The accuracy and breadth of our recommendation prototypes can be further improved. We currently employ a simple statistical distance measure for unimodal data (*i.e.,* considering each signal independently). There is a great opportunity for future work to improve this aspect by developing more sophisticated methods of learning from usage data. Given a large enough sample, it may even be possible to learn retroactively from sensor streams collected from people who have turned on a feature.

Our approach to matching signals to detection strategies was largely manual. In some cases, we believe the mappings are fairly straightforward extensions of the accessibility feature specification, especially for the *Near-Miss* category that is explicitly defined this way. However, other categories, such as *Sequence* or *Statistical*, require generating hypotheses about differences exhibited by people who could benefit from the feature, collecting data to validate this hypothesis, and finally building a recommendation strategy based on that. Future work could investigate automated data-driven methods of identifying promising accessibility signals from usage patterns. In pursuing this, it is important to collect and analyze this data in a privacy-preserving manner. The *Statistical* signals rely on detecting differences from a collected baseline, and so developers of recommenders using this approach should be cognizant of where that baseline is collected from and be aware that some people may differ from the baseline for a reason other than needing the accessibility feature. Our framework does not provide a pattern for recommending all features that can be naively applied; future systems based on detecting and using accessibility signals should continue to rely on the intuition of designers, feedback from potential users, and iterative development and evaluation.

Another area to more thoroughly explore is how to surface accessibility recommendations. In our usage study, we presented users with recommendations after they completed their tasks, but recommendations can also be presented *in situ*. Notifications may be the most direct way of capturing attention and displaying information to users, but they can be disruptive or annoying [320]. Initiatives from Apple [4] and Google [2] have focused on limiting interruptions from notifications. The aforementioned “Sticky Keys” notification on Microsoft Windows has arguably been successful in getting people to know about the feature, but numerous web-based articles are devoted to turning off that notification (some fast-paced video games also often involve pressing modifier keys repeatedly and quickly). Less obtrusive ways of surfacing recommendations could include simply ranking the features higher in the accessibility menu, or including them in a non-intrusive but clearly visible place (*e.g.,* the lock screen) [11].

As a part of our exploration, we briefly explored different ways recommendations might be surfaced in a mobile operating system. While we expect more obtrusive notifications like pop-ups to be more likely attended to by users, they might also be more likely to disturb or annoy users[2]. Ultimately, there are many design decisions to be made, such as the wording and style used in the recommendation. In our prototypes, we specifically avoided relating the recommended feature to any underlying condition, ability, or cause. For example, we wrote recommendations like, *“Did*

[2] Early in development, we also implemented a modal dialog box as an alert, but pilot studies guided us to less obtrusive designs.

you know you can adjust the font size?" rather than *"It looks like you're having trouble seeing the screen."* We suspect that the preferred style and obtrusiveness of notifications may depend on the accuracy and timing of the recommendation, the likelihood that the user will follow the advice, and the realized benefit to them if they adopt the feature (*i.e.*, the classic expected utility problem in mixed-initiative interaction [141]). Ultimately, we believe the best strategy and frequency for surfacing recommendations will be feature and context dependent. We leave it to future work to empirically determine the best strategy for each feature and use case.

3.8 Conclusion

A large number of accessibility features have been developed for smartphone platforms. Our survey with 100 participants demonstrates that relatively few know about these features or how they might benefit from them. In this paper, we present our framework for recommending accessibility, outlining useful signals and detection methods using them to recommend accessibility features on the smartphone platform. We categorize 48 features on iOS in terms of how those features might be recommended to participants, and provide a number of example recommenders. We develop four prototypes, three of which we initialized in a baseline study with 10 participants. We then use our recommenders with another population of 20 participants to better understand their potential feasibility and utility. With so many great accessibility features being developed, we believe it is important to direct some of our research focus to recommending these accessibility features to those who might benefit from them. Our work provides a roadmap for researchers and developers to make progress in this important area.

4

WebUI: A Dataset for Enhancing Visual UI Understanding with Web Semantics

4.1 Preamble

Content in this chapter is based on a conference paper that was published at CHI 2023 [330].

In the previous chapter, I showed that under certain circumstances (*i.e.,* when semantic information is available), useful transformations, such as built-in feature configurations, can be applied at runtime to better adapt existing UIs to user context. Examples of these semantics include rendering metadata (*e.g.,* the accessibility hierarchy of mobile apps) or complete source code (*e.g.,* some types of web UIs), which describe the information and functionality presented by the UI. Operating system features and assistive technologies are some examples of mechanisms that depend on this metadata to function. However, these application semantics are often unavailable on many types of graphical user interfaces (GUIs) that are constructed using inaccessible toolkits. To this end, some prior work has focused on building datasets for training machine learning models that can predict semantic metadata (which is not always provided by the developer) from an app's screenshot (which is always available for GUIs). The purpose of the WebUI project is to build a dataset of UIs where visual appearance (*i.e,* screenshots) are associated with app semantics, with the goal of training models that can relate and generate one representation from another.

Modeling user interfaces (UIs) from visual information allows systems to make inferences about the functionality and semantics needed to support use cases in accessibility, app automation, and testing. Current datasets for training machine learning models are limited in size due to the costly and time-consuming process of manually collecting and annotating UIs. We crawled the web to construct WebUI, a large dataset of 400,000 rendered web pages associated with automatically extracted metadata. We analyze the composition of WebUI and show that while automatically extracted data is noisy, most examples meet basic criteria for visual UI modeling. We applied several strategies for incorporating semantics found in web pages to increase the performance of visual UI understanding models in the mobile domain, where less labeled data is available: *(i)* element detection, *(ii)* screen classification and *(iii)* screen similarity.

4.2 Introduction

Computational modeling of user interfaces (UIs) allows us to understand design decisions [80, 170], improve their accessibility [347], and automate their usage [54, 191, 196]. Often, these systems must interact with UIs in environments with incomplete or missing metadata (*e.g.,* mobile apps authored with inaccessible UI toolkits). This presents many challenges since it necessitates that they reliably identify and reason about the functionality of the UI to support downstream applications. Visual modeling of UIs, which has shown to be a promising solution, predicts information directly from a screenshot using machine learning models and introduces no additional dependencies.

Building the datasets needed to train accurate visual models involves collecting a large number of screenshots paired with their underlying semantic or structural representations. Recent efforts to collect datasets [80, 347] for data-driven modeling have focused on mobile apps, which are typically manually crawled and annotated by crowdworkers since they are often difficult to automate. This process is both time-consuming and expensive — prior work has estimated that collecting a dataset of 72,000 app screens from 10,000 apps took 5 months and cost $20,000 [80]. Because of this, datasets for visual UI modeling are limited in size and can be prohibitively expensive to keep updated.

The web presents a possible solution to UI data scarcity since web pages are a promising source of data to bootstrap and enhance visual UI understanding. In contrast to mobile UIs, web UIs (*i.e.,* web pages) are much easier to crawl since they are authored in a unified parsable language (*i.e.,* HTML) that typically exposes semantics (*e.g.,* links and listeners) necessary for automated navigation. The same web page can also be viewed in many different viewports and display settings, which makes it possible to collect a large dataset of UIs rendered on a variety of devices (*e.g.,* a smartphone or tablet). In addition, web browsers offer several facilities to extract visual, semantic, and stylistic information programmatically. In particular, web conventions, such as the semantic HTML and the ARIA initiatives, while not always adopted, constitute a large, if potentially noisy, source of annotations for UI elements. Finally, the web offers a virtually unlimited supply of data and has already been employed as a data source for large-scale machine learning [117, 335, 337]. We explore the possibility of automatically collecting and labeling a large dataset of web UIs to support visual UI modeling in other domains (e.g., mobile). Compared to previous web datasets [170], our dataset is much larger, more recent, and contains semantic information needed to support common visual UI understanding tasks.

In this paper, we show that a large dataset of automatically collected web pages can improve the performance of visual UI Understanding models through transfer learning techniques, and we verify this phenomenon for three tasks. We first describe the platform that we built to crawl websites automatically and scrape relevant visual, semantic, and style data. Our crawler visited a total of approximately 400,000 web pages using different simulated devices. WebUI, the resulting dataset is an order of magnitude larger than other publicly available datasets [170]. Next, we analyzed our dataset's composition and estimated data quality using several automated metrics: *(i)* element size, *(ii)* element occlusion, and *(iii)* layout responsiveness. We found that most websites met basic criteria for visual UI modeling. Finally, we propose a framework for incorporating web semantics to enhance the performance of existing visual UI understanding approaches. We apply it to three tasks in the literature: *(i)* element detection, *(ii)* screen classification and *(iii)* video screen similarity and show that incorporating web data improves performance in other target domains, even when labels are unavailable.

To summarize, our paper makes the following contributions:

1. The WebUI dataset, which consists of 400,000 web pages each accessed with multiple simulated devices. We collected WebUI using automated web crawling and automatically associated web pages with visual, semantic, and stylistic information that can generalize to UIs of other platforms.
2. An analyis of the composition and quality of examples in WebUI for visual UI modeling in terms of *(i)* element size, *(ii)* element occlusion, and *(iii)* website layout responsiveness.
3. A demonstration of the usefulness of the WebUI dataset through three applications from the literature: *(i)* element detection, *(ii)* screen classification and *(iii)* screen similarity. We show that incorporating web data can lead to performance improvements when used in a *transfer learning* setting, and we verified its improvement for our three tasks. We envision that similar approaches can be used for other tasks common in visual UI understanding. Furthermore, we show that models trained on only web data can often be directly applied to other domains (*e.g.,* Android app screens).

All code, models, and data will be released to the public to encourage further research in this area.

4.3 WebUI Dataset

We introduce the WebUI dataset, which we construct and release to support UI modeling. The WebUI dataset is composed of 400,000 web pages automatically crawled from the web. We stored screenshots and corresponding metadata from the browser engine, which serve as annotations of UI element semantics. Because the collection process is highly automated, our final dataset is an order of magnitude larger than other publicly available ones (Figure 4.4) and can be more easily updated over time.

In this section, we give an overview of our web crawling architecture, analyze the composition of our dataset, and provide evidence that it can support visual UI modeling for other platforms.

4.3.1 Web UI Crawler

Crawling Architecture

To collect our dataset, we implemented a parallelizable cloud-based web crawler. Our crawler consists of *(i)* a crawling coordinator server that keeps track of visited and queued URLs, *(ii)* a

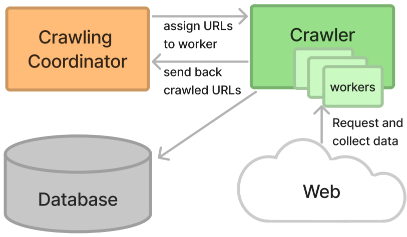


Figure 4.1: Overview of our crawling architecture. A *crawling coordinator* contains a queue of URLs to crawl and assigns them to workers in a *crawler pool*. Workers asynchronously process URLs by visiting them in a automated browser, scraping relevant metadata, then uploading them to a cloud database.

pool of crawler workers that scrapes URLs using a headless browser, and *(iii)* a database service that stores uploaded artifacts from the workers. The crawler worker is implemented using a headless framework [20] for interfacing with the Chrome browser. Each crawler worker repeatedly requests a URL from the coordinator server, which keeps global data structures for visited and upcoming URLs. The crawler worker includes some simple heuristics to automatically dismiss certain types of popups (*e.g.,* GDPR cookie warnings) to help it access page content.

We seeded our coordinator using a list of websites that we hypothesized would lead to diverse examples of web pages (e.g., link aggregation websites and design blogs) and ones that we expected to have high-quality accessibility metadata (*e.g.,* government websites). A full list of our seed websites can be found in the supplementary materials.

We explored several crawling policies and eventually settled on one that encourages diverse exploration by inversely weighting the probability of visiting a URL by its similarity to the visited set. For example, if the crawler previously visited `http://example.com/user/alpha`, it would be less likely to subsequently visit `http://example.com/user/beta`. We set a minimum probability so that it is possible to re-visit links to support additional types of analysis (e.g., temporal changes). The coordinator organizes upcoming (i.e., queued) URLs by their hostname, *(i)* selects a hostname randomly with uniform probability, and then *(ii)* selects a URL using its assigned probability. Empirically, we found this technique to be effective at avoiding crawler traps, which are websites that cause automated crawlers to get stuck in endless loops navigating within the same site.

Data Collected from a Web Page

We used a pool of crawler workers to crawl web pages in parallel, and we visited each URL with multiple simulated devices. We collected several types of semantic information by querying the rendering and accessibility engine. We set a timeout limit of 6 minutes for each URL, so some web pages were not visited by all simulated devices.

Simulated Devices. We sampled each web page with 6 simulated devices: 4 of the most common desktop resolutions [18], a tablet, and a mobile phone. Devices are simulated by setting

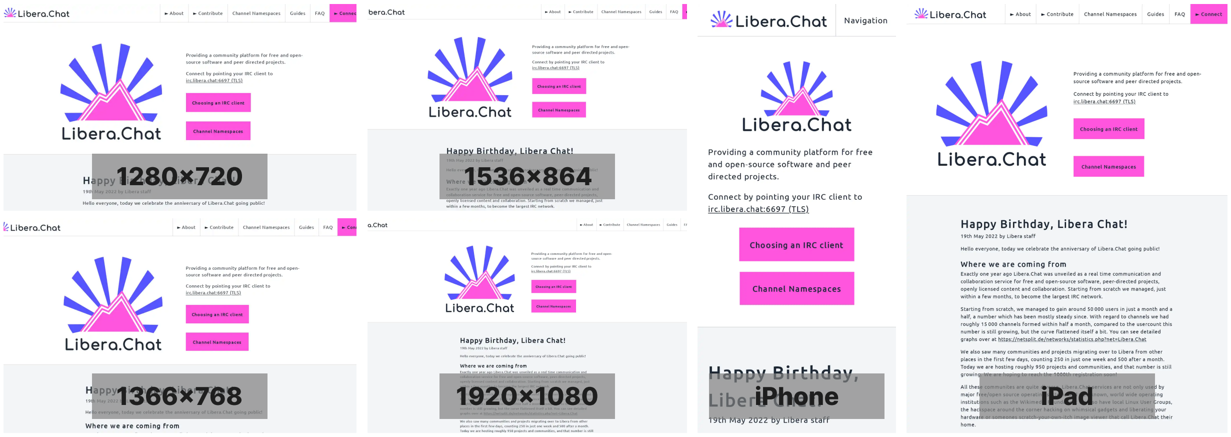


Figure 4.2: Screenshots from a web page accessed using 6 different devices: 4 desktop resolutions, a smartphone, and a tablet. By requesting a responsive web page at different resolutions, we induce several layout variations (*e.g.,* navigation and hero button).

the browser window resolution and user agent to match the goal device, both of which may affect the page's content and rendering.

Screenshots. Our crawler worker captured two types of screenshots (*i.e.,* visual data) from websites. We captured a viewport screenshot, with fixed image dimensions, and a full-page screenshot, with variable height. Images were saved using lossy compression to save storage. While compression can introduce some artifacts, previous work [89] suggests that the effect on deep learning model performance is minimal.

Accessibility Tree. We used a browser automation library to query Chrome's developer tools to retrieve an accessibility tree for each page [19]. The accessibility tree is a tree-based representation of a web page that is shown to assistive technology, such as screen readers. The tree contains accessibility objects, which usually correspond to UI elements and can be queried for properties (*e.g.,* clickability, headings).

Compared to the DOM tree, the accessibility tree is simplified by removing redundant nodes (e.g., ¡div¿ tags that are only used for styling) and automatically populated with semantic information via associated ARIA attributes or inferred from the node's contents. The browser generates the accessibility tree using a combination of HTML tags, ARIA attributes, and event listeners (*e.g.,* click handlers) to create a more consistent semantic representation of the UI. For instance, there are multiple ways to create a button (*e.g.,* a styled `div`) and the accessibility tree is intended to unify all of these to a single `button` tag.

Layout and Computed Style. For each element in the accessibility tree, we stored layout information from the rendering engine. Specifically, we retrieved 4 bounding boxes relevant to the "box model": *(i)* the content bounding box, *(ii)* the padding bounding box, *(iii)* the border bounding box, and *(iv)* the margin bounding box. Each element was also associated with its computed style information, which included font size, background color and other CSS properties.

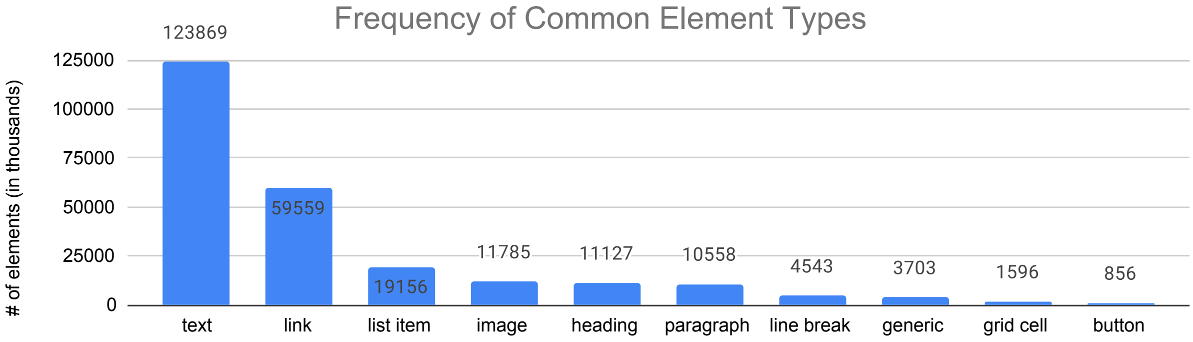


Figure 4.3: 10 most common element types in the WebUI dataset. Element types are based on automatically computed roles, which are not mutually exclusive. Text is the most common type, but many types offer semantic information about what text is used for *e.g,* a heading, paragraph or link.

4.3.2 Dataset Composition

The WebUI dataset contains 400K web UIs captured over a period of 3 months and cost about $500 to crawl. We grouped web pages together by their domain name, then generated training (70%), validation (10%), and testing (20%) splits. This ensured that similar pages from the same website must appear in the same split. We created four versions of the training dataset. Three of these splits were generated by randomly sampling a subset of the training split: Web-7k, Web-70k, Web-350k. We chose 70k as a baseline size, since it is approximately the size of existing UI datasets [80, 347]. We also generated an additional split (Web-7k-Resampled) to provide a small, higher quality split for experimentation. Web-7k-Resampled was generated using a class-balancing sampling technique, and we removed screens with possible visual defects (*e.g.,* very small, occluded, or invisible elements). More information about how this set was generated can be found in the appendix. The validation and test split was always kept the same.

Comparison to Existing Datasets

WebUI is an order of magnitude larger than existing datasets used for UI understanding (Figure 4.4) and provides rich semantic and style information not found in mobile datasets. WebUI focuses on the static properties of web pages and does not store page loading times or element animations.

We analyzed the makeup of web UIs and compared them to mobile UIs. The distribution of UI types (*e.g.* Login, News, Search) in WebUI are also likely to be different than mobile data, since many web pages are primarily hypertext documents. We extracted elements from the accessibility tree and categorized them using their computed accessibility role and the role of any singleton parents. For example, a clickable image is created in HTML by surrounding an image (¡img¿) element with an anchor element (¡a¿). Thus, it is possible for elements to be assigned to multiple classes. Figure 4.3 shows the frequency of element types in our dataset. Similar to prior work [347], we find that text is the most common element in our dataset. However, we find

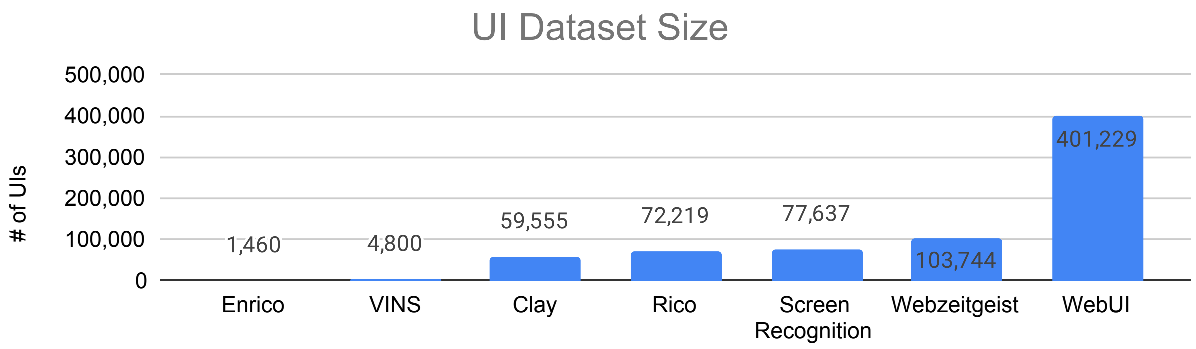


Figure 4.4: Comparison of WebUI to existing UI datasets. WebUI contains nearly 400,000 web pages and is nearly one order of magnitude larger than existing datasets available for download (Enrico, VINS, Clay, Rico). Each web page also contains multiple screenshots captured using 6 simulated devices.

limited overlap between the rest of the label set, possibly due to the nature of web data and the mutually exclusive nature of existing label sets. On average, there were 60 elements on a web UI, 30 of which were visible in the viewport. This is more than the number of elements on mobile app screens, which prior work estimated to be around 25 per screen, although this may in part be due to differences in segmentation (*e.g.,* a single Rich Text Field on Android can contain differently formatted text while on HTML they would broken up into different tags). On average, there were also more clickable elements per web page (20 on web pages vs 15 “interactable” elements on Android apps), likely due to the prevalence of hyperlinks on the web.

Dataset Quality

Compared to manually labeled examples, automatically extracted annotations can contain errors that impact modeling performance. We conducted an analysis on a small, randomly sampled data from our dataset (1000 web pages). While there are numerous possible defects, we focus on three that we believe are most relevant to data quality: *(i)* element size, *(ii)* element occlusion, and *(iii)* website responsiveness. Our analysis is primarily focused on quantifying possible defects but not reparing them. Previous work [185, 281] has explored automated methods for correcting mismatched labels and occluded elements, and we expect the overall quality of WebUI could be improved if these were applied..

Element Size. Element size refers to the dimensions of an annotated object in an image. For example, if a bounding box annotation surrounds an object that is too small relative to the image resolution, it may be difficult for a model to identify the object. The average area of bounding boxes in our data is approximately $14000px^2$, but this may have been influenced by short segments of text. The Web Content Accessibility Guidelines (WCAG) guideline for target size also recommends that interactable elements have a minimum size of 44 by 44 pixels, so that they can be easily selected by users. In our dataset, one third of interactable elements (*e.g.,* elements tagged as links or button) were smaller than this threshold.

Element Occlusion. Element occlusion occurs when one object partially or completely covers another in a screenshot. Occluded elements are detrimental to visual modeling since they may represent targets that can be impossible to predict correctly. We quantified the occlusion rate by counting the number of screens with overlapping leaf elements. We found that 18% of screens in our sampled split contained overlapping leaf elements. However, of the overlapping elements, only a third of them were occluded by more than 20% of their total area.

Responsive Websites. Website responsiveness relates to how well a web page adapts to different screen viewports. Since we simulated multiple devices for each web page, responsive websites are likely to produce more variation in their layouts than unresponsive ones. To measure responsiveness, we automatically computed metrics included in the Chrome Lighthouse tool for estimating layout responsiveness: *(i)* responsiveness of content width to window size and *(ii)* the use of a viewport meta tag, which is needed for proper mobile rendering. From our analysis we found that 70% and 80% of processed web pages met the first, and second criteria, respectively.

In summary, our analysis suggests that most web pages in our dataset meet some basic requirements for visual UI modeling. Given the reliance of our data collection on extracted accessibility metadata, we expect high quality examples to adhere to good accessibility practices, such as those outlined by WCAG. However, considering the inaccessibility of the web and that many criteria are difficult to verify automatically, we also expect many web pages to violate some of these criteria. There are other desirable properties for dataset quality that we did not check, *e.g.,* the accurate use of semantic HTML tags, ARIA tags, and tightness of element bounding boxes. These properties were harder to verify automatically, since they require knowledge of developer intention and associated tasks. In our analysis, we only attempt to identify possible defects, and we did not attempt to remove or repair samples. This could be a direction for future work to improve dataset quality [57, 185].

4.4 Transferring Semantics from Web Data

We hypothesized that web data is similar and relevant to modeling other types of UIs from their pixels. In this paper, we are specifically interested in the mobile domain, as mobile apps often lack metadata and can only be reliably understood from their visual appearance. In many cases, manually-annotated mobile datasets are small, and in some cases, labels are completely unavailable. We used transfer learning to apply our dataset to three existing tasks in the UI understanding literature: *(i)* element detection, *(ii)* screen classification, and *(iii)* screen similarity. Table 4.1 shows downstream applications where UI understanding tasks can benefit from web data. Because each task contains different constraints (*e.g.,* presence of labeled target data) it is difficult to apply a single strategy to serve all use-cases. For example, inductive transfer learning typically requires labels in both the pre-training and fine-tuning phase is impossible to apply to a setting where target labels are unavailable (*e.g.,* screen similarity). We expect our three transfer learning strategies to be applicable to most future use-cases, since they span all combinations of labeled data availability (Table 4.1).

Table 4.1: Table of strategies for transferring semantics from web pages to other types of UIs. We explored scenarios where labeled data is missing in either domain by applying three strategies: *(i)* finetuning, *(ii)* semi-supervised learning, and *(iii)* domain adaptation.

Approach	Finetuning	Semi-supervised Learning	Domain Adaptation
Application	Element Detection	Screen Classification	Screen Similarity
Web (Source)	Y	N	Y
Mobile (Target)	Y	Y	N

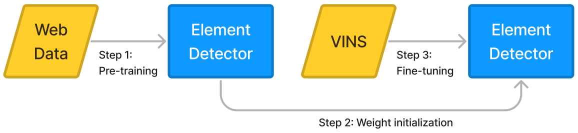


Figure 4.5: We applied inductive transfer learning to improve the performance of a element detection model. First, we pre-trained the model on web pages to predict the location of nodes in the accessibility tree. Then, we used the weights of the web model to initialize the downstream model. Finally, we fine-tuned the downstream model on a smaller dataset consisting of mobile app screens.

4.4.1 Element Detection

Element detection requires a machine learning model to identify the locations and types of UI elements from a screenshot. Often these models are based on object detection frameworks.

Element detection is an example of a task where labeled data is available in both the source and target domain (albeit fewer examples of mobile screens), so it is possible to employ inductive transfer learning. The WebUI dataset contains the locations of elements that we scraped from the website accessibility tree. Element types are inferred from the HTML tags and the ARIA labels [19]. We show that this training strategy results in improvements to element detection performance.

Model Implementation

We primarily followed the details provided by VINS [53] to implement our element detection model. The VINS dataset, which we used for training, is composed of 4800 annotated UI screenshots from various sources such as design wireframes, Android apps, and iOS apps. Since the authors did not release official data splits, we randomly partitioned the data into training (70%), validation (15%), and testing (15%) sets. This specific split ratio was chosen since it has been used in other UI modeling work [326]. The paper identifies 11 primary UI component classes; however the released raw dataset includes a total of 22 class labels. For the extraneous labels, we either tried to merge them with the 11 primary labels (*e.g.,* "Remember Me" merged with "Check Box") or assigned them to an "Other" class (*e.g.,* "Map") if no good fit was found. Instead of the SSD object detection model [214] used by VINS, we opted to start from the more recent FCOS model architecture [305], since we found it was easier to modify to support multi-label training. Previous element detection work [53, 63, 347] trained models to assign one class label (*e.g.,* Button, Text field) to each detected element in the screenshot. To take advantage of multiple, nested definitions of web elements in our dataset, we trained the object detection model to predict multiple labels for each bounding box.

Figure 4.5 illustrates the overall training process. In the pre-training phase, the element detection model is trained on a split of the WebUI dataset. Due to cost and time constraints, we trained all element detection models for a maximum of 5 days. We also used early stopping on the validation metric to reduce the chance of overfitting. Afterwards, a specific part of the model was re-initialized (the object classification head) to match the number of classes in the VINS dataset before it was fine-tuned. We found it difficult to modify the original SSD architecture to support the multi-label pre-training, so we only followed the original training from scratch procedure described in the paper as a baseline.

Results

Table 4.2 shows the performance of each model configuration on the VINS test set, and we show that our updated configurations lead to significant performance improvements. Our primary performance metric for this task was the mean average precision (mAP), which is a standard metric used for object detection models that takes into the accuracy of bounding box location (*i.e.,* how closely the predicted box overlaps with ground truth) and classification (prediction of object type). The mAP score is calculated by computing an individual average precision (AP) score for each possible element class (*e.g.,* Text, Check Box), which represents the object detector's accuracy in detecting each object class. The AP scores are averaged to produce the mAP score. We calculated the mAP score over classes that could be mapped to the original label set in the paper [53] *i.e.,* we excluded the "Other" class where there was no clear mapping to the original set. We calculated the un-weighted mean between class APs, which assigns equal importance to common and rare element types. Our best model configuration performed 0.14 better than the baseline in terms of mAP score. While the largest source of improvement over the baseline configuration (SSD) came from the updated FCOS model architecture, our fine-tuning procedure contributed to gains as well. Specifically, we note that pre-training with more examples led to better performance (around 0.04 mAP). Depending on the downstream application of the element detection model, this improvement could lead to better user experience but would require further validation. For example, a screen reader [347] does not require tight bounding boxes; however, it would benefit from detecting more (small) elements on the screen. Query-based design search [53] could also retrieve more relevant examples.

Although we followed the original training procedure as closely as possible, we were unable to reach the mAP score reported in the original VINS paper. This can be attributed to *(i)* our use of different randomized splits and *(ii)* differences in mappings between class labels from the raw data to the 11 primary classes, which were not provided in the previously released code. Nevertheless, since we used the same splits and class mappings across all of our model configurations, we expect the relative performance improvements to be consistent.

We also investigated the zero-shot performance of element detectors trained only on web data (*i.e.,* without fine-tuning). It is difficult to compute performance quantitatively, since the label sets between the web and mobile datasets do not directly overlap. However, we provide

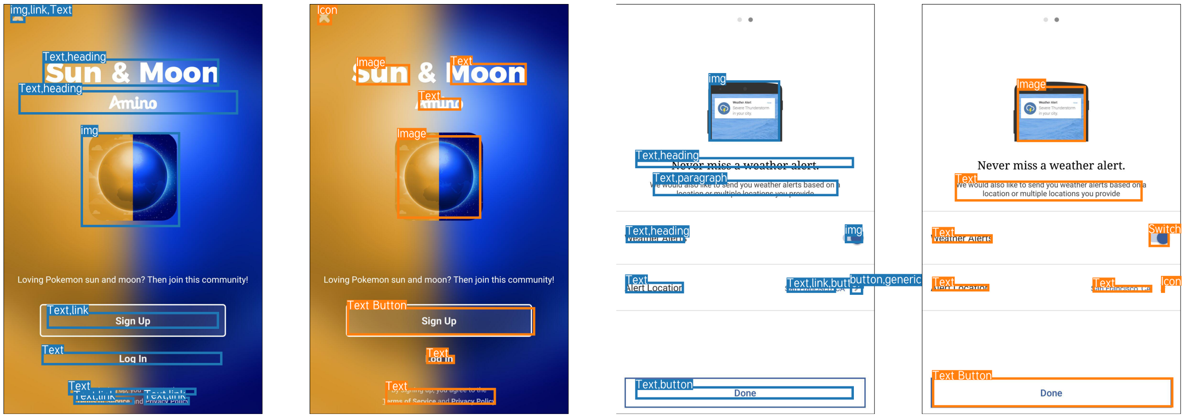


Figure 4.6: Output of our element detection models run on two app screens. In many cases, detections from our web-only model (Blue) coincide with ones from our fine-tuned model (Orange), which suggests some zero-shot transfer capabilities. Predicted tags from the web-only model also provide additional metadata corresponding to clickability (`link`) and heading prediction (`heading`); however, the predicted bounding boxes are often less tight than the fine-tuned model.

Table 4.2: Element detection performance (11 object classes) for different model configurations. Pre-training on more web screens led to better performance on mobile screens after fine-tuning.

Model Configuration	mAP
SSD (Random Init.)	0.6737
FCOS (Random Init.)	0.7739
FCOS (Pre-trained on Web7k)	0.7877
FCOS (Pre-trained on Web7k-Resampled)	0.7961
FCOS (Pre-trained on Web70k)	0.7921
FCOS (Pre-trained on Web350k)	0.8115

qualitative evidence that zero-shot learning could be successful. Figure 4.6 shows the output of a web model when run on mobile app screens from Rico. We conducted minimal preprocessing, such as cropping out the Android system notification bar and the navigation soft buttons. In many cases, the web analogs of mobile text and image elements are detected accurately, which suggests that some element classes have consistent appearance across platforms. Interestingly, some web classes such as links and headings are also detected in the image, which could be used to infer new semantics such as clickability [294] and navigation landmarks.

4.4.2 Screen Classification

Classifying screen type or functionality from a screenshot can be useful for design analysis and automation. Previously, small amounts of data have been collected and annotated for this purpose. Enrico [181] is an example of a dataset (1460 samples, subset of Rico [80]) where each screenshot is assigned to one of 20 mutually-exclusive design categories. Because of the dataset's

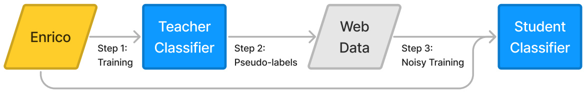


Figure 4.7: We applied semi-supervised learning to boost screen classification performance using unlabeled web data. First, a teacher classifier is trained using a "gold" dataset of labeled mobile screens. Then, the teacher classifier is used to generate a "silver" dataset of pseudo-labels by running it on a large, unlabeled data source (*e.g.,* web data). Finally, the "gold" and "silver" datasets are combined when training a student classifier, which is larger and regularized with noise to improve generalization. This process can be repeated; however, we only perform one iteration.

small size, it is challenging to train accurate deep learning classification models. While our web dataset is large, it also does not have the screen-type annotations, and thus it is not possible to employ the same pre-training strategy that was used for element detection.

Instead, we applied a semi-supervised learning technique known as self-training [59]. Self-training is a process that improves model performance by iteratively labeling and re-training on a large source of unlabeled data. We investigated the effects of using WebUI as the unlabeled dataset and show that doing so improves overall screen classification accuracy.

Model Implementation

Figure 4.7 shows our procedure for incorporating WebUI data into our model training via self-training.

First, we trained screen classifier based on the VGG-16 architecture with batch normalization and dropout [287], as described by the Enrico paper [181]. Since official training, validation, and testing splits were not provided, we randomly generated our own (70%/15%/15%). This model was trained only on data from the Enrico training split and served as the *teacher classifier*. Next, the teacher model was used to generate "soft" pseudo-labels for screenshots in the WebUI dataset, where each sample was mapped to a vector containing probabilities for each class. We followed the procedure used by Yalniz et al. [337] to keep only the top K most confident labels for each class. To select K, we first randomly sampled a small subset of 1000 web pages from our dataset and performed a parameter search to find the optimal value. Based on our experiments, we found that a value of 10% of the total dataset size led to good performance (*e.g.,* we set K=700 for the Web-7k split). Finally, we trained a *student classifier* on a combination of the original and automatically generated labels. We employed a specific type of self-training known as Noisy Student Training [335], which involves injecting noise into the student model's training process so that it becomes more robust. Two types of noise are used in this process: *(i)* input noise, which is implemented via random data augmentation techniques and *(ii)* model noise, which is implemented with dropout [291] and stochastic depth [148]. Because stochastic depth can only be applied to model architectures with residual blocks, we used an architecture based on ResNet-50 [134] instead of VGG-16.

Table 4.3: Classification accuracy (across 20 classes) for different configurations of our screen classification model. Increasing the amount of data used with our semi-supervised learning method led to increased accuracy.

Model Configuration	Accuracy
VGG-16	0.4737
Noisy ResNet-50	0.4649
Noisy ResNet-50 (Rico)	0.4956
Noisy ResNet-50 (Web7k)	0.4864
Noisy ResNet-50 (Web7k-Resampled)	0.4868
Noisy ResNet-50 (Web70k)	0.5175
Noisy ResNet-50 (Web350k)	0.5263

Results

Overall, we found that applying self-training to incorporate additional unlabeled data led to consistent performance improvements (Table 4.3). The best classifier using WebUI data was 5% more accurate than the baseline model, which was only trained with the Enrico dataset. Our baseline VGG-16 model performed considerably worse than the originally reported results [181] but achieved similar accuracy to another reproduction of the work [203]. The performance difference could be attributed to differences in randomized splits. Since we used the same splits across all conditions, we expect relative performance differences to be consistent. To investigate the effects of using a new model architecture, we also trained a Noisy ResNet-50 (architecture used by the student model) on the Enrico dataset. The resulting classifier performed relatively poorly (worse than the baseline model), since the modifications introduced (dropout and stochastic depth) require more data to train effectively.

The primary source of improvement stems from the inclusion of additional unlabeled data during the training process, which led to a more generalizable student model. We observed that the small size of the Enrico dataset (1460 samples) quickly led to overfitting during training and limited overall performance. Semi-supervised learning techniques, such as self-training, allow training on a much larger number of examples. We found that model accuracy improved when we incorporated more unlabeled examples, both from WebUI and Rico.

4.4.3 Screen Similarity

Identifying variations within the same screen and detecting transitions to new screens are useful for replaying user interaction traces, processing bug reports [74], and automated app testing [199, 201]. To model these properties and understand how multiple screens from an application relate to each other, previous work [100, 201] has sought to differentiate between distinct UIs and variations of the same UI. For example, the same checkout screen may appear different based on the number and types of products added to the cart. Common screen interactions such as scrolling and interaction with expandable widgets (*e.g.,* menus, dialogs, keyboards, and notifications) may also alter the visual appearance of a screen. Visual prediction reduces system

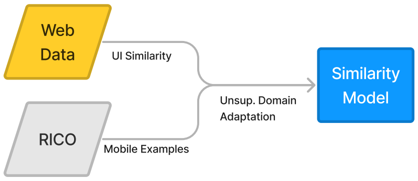


Figure 4.8: We used unsupervised domain adaptation (UDA) to train a screen similarity model that predicts relationships between pairs of web pages and mobile app screens. The training uses web data to learn similarity between screenshots using their associated URLs. Unlabeled data from Rico is used to train an *domain-adversarial* network, which guides the main model to learn features that transferrable from web pages to mobile screens.

reliance on accessibility metadata, which may be missing or incomplete, and further extends the applications of these models, as they can process video recordings of user interactions (*e.g.,* reproducing bug reports) [40, 74].

Previous work [100] opted to manually annotate a dataset of more than one thousand iPhone applications that were manually "crawled" by crowdworkers; however, the dataset was not released to the public. As a weak source of annotation, we used web page URLs to automatically label page relations. Since no labeled data is available in the mobile domain, we employed domain-adversarial network training [116], a type of unsupervised domain adaptation (UDA), to encourage the model to learn transferrable features from the web domain that might apply to the mobile domain. Note that while it is possible to apply the semi-supervised learning strategy (which was used for the screen classification task) in reverse, it may be less effective, since the unlabeled dataset (mobile UIs) is smaller than the labeled dataset.

Model Implementation

We followed previous work [100] and used a ResNet-18 [134] model trained as a siamese network [131]. The siamese network uses the same model to encode two inputs, then compares them in feature space (*i.e.,* their embeddings) to decide if they are different variations of the same UI screen. Our approach is different from the method proposed by previous work [74], which applies random data augmentations (*e.g.,* blurring, rotation, translation) to screenshots to create *same-screen* pairs. Instead, we randomly sampled pairs of screenshots from our web data for training, with balanced probability for same-screen and new-screen pairs. Same-screen pairs were generated by finding screenshots with the same URL but accessed at different times or simulating page scrolls on a full-page screen capture by sliding a window vertically along the image. Note that occasionally, simulated page scrolls and accessing the same web page at different times still produced identical or nearly identical screenshots, so in our test set, we filtered these out using perceptual hashing. Different-screen pairs were generated both by sampling screenshots from within the same domain but with different URL path, and by sampling screenshots from other domains.

The domain-adversarial training process seeks to simultaneously accomplish two objectives: *(i)* learn an embedding space where two screenshots are from the same screen if their distance is less than a threshold, and *(ii)* learn an encoding function that applies to both the web and mobile

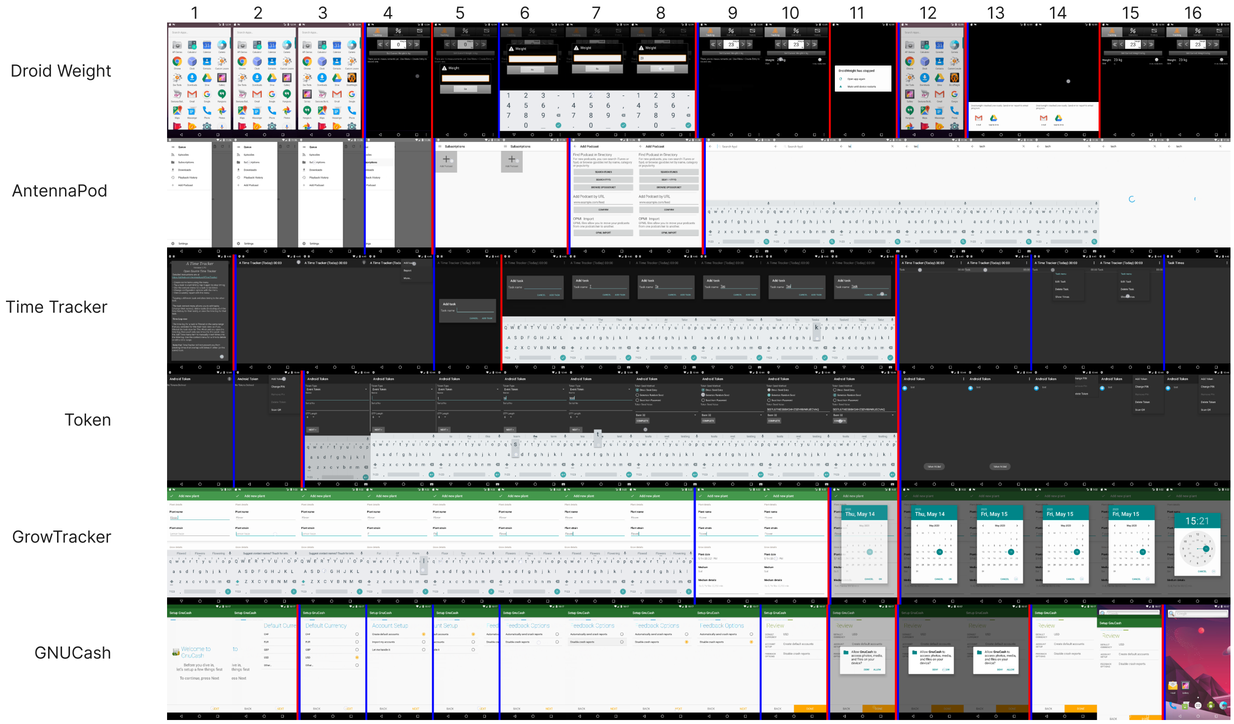


Figure 4.9: Examples of interaction videos segmented by our best models trained with UDA (Red) and without UDA (Blue). Videos are sampled at 1 fps. The output of both models contain errors, however, we found that the adapted UDA model generally produced better segmentations. Common errors include oversegmentation due to app dialogs and soft keyboards, which do not occur in the WebUI dataset.

domains. The first objective is related to the primary task of distinguishing same-screen pairs from new-screen pairs and is achieved with a pairwise margin-based loss [100]. The second objective aims to align the feature distributions of the two domains by *maximizing* the error rate of a domain classifier, which is a network that tries to classify whether a sample is from a web or mobile UI. For this task, we used only web page screenshots captured on simulated smartphones, to make the domain classification objective more challenging.

Results

Since one of the assumptions of our problem is that labeled examples of same-screen and new-screen pairs are unavailable for mobile apps, we used two alternative methods to evaluate our screen similarity model: *(i)* quantitative evaluation on labeled pairs of web screens and *(ii)* qualitative evaluation on a set of unlabeled Android interaction videos.

Table 4.4 shows the quantitative performance of our models evaluated on pairs of web pages from our dataset. Overall, training with more data led to significantly better performance, an increase of over 20%. The inclusion of a domain adaptation objective sometimes led to a slight drop in classification performance since it introduces additional constraints in the learning process. We qualitatively evaluated our model's performance characteristics on mobile screens by

Table 4.4: Classification performance (*same-screen* vs *new-screen*) of our screen similarity models evaluated on pairs of screens from our web data. Performance increased when the model was trained on more data and slightly decreased when trained with the UDA objective.

Model Configuration	F1-Score
ResNet-18 (Web7k)	0.7097
ResNet-18 UDA (Web7k)	0.7184
ResNet-18 (Web7k-Resampled)	0.7368
ResNet-18 UDA (Web7k-Resampled)	0.7191
ResNet-18 (Web70k)	0.8222
ResNet-18 UDA (Web70k)	0.8193
ResNet-18 (Web350k)	0.9630
ResNet-18 UDA (Web350k)	0.9500

using them to segment videos of mobile app interaction. We used a dataset of screen recordings of bug reproductions [74] for 6 open-source Android apps and applied our model by sequentially sampling frames from the video and evaluating whether a new screen was reached. Note our sampling process differs from other previous work [54, 80] that segmented crawls at recording time using accessibility metadata, because we do not have accessibility metadata corresponding to the previously collected recordings used in our analysis. Figure 4.9 shows an example of a usage video processed by our model. While the web model was effective detecting some types of transitions that occurred in mobile apps, it was less effective at others, such as software keyboards and dialogs, which do not occur frequently in the WebUI dataset. We include more model-generated segmentations of the bug reproduction dataset in supplementary material.

In this work, we applied *unsupervised* domain adaptation, which does not require any labels from the target domain. Other domain adaptation strategies exist, and some are able to incorporate small amounts of labeled data, which we expect could improve the accuracy of our model by contributing transition types unique to mobile apps.

4.5 Discussion

4.5.1 Performance Impact of Web Data

Empirically, we showed that automatically crawled and annotated web pages, like those available in WebUI, can effectively support common visual modeling tasks for other domains (*e.g.,* mobile apps) through transfer learning strategies. In cases where a small amount of labeled mobile data was available, as in element detection and screen classification, incorporating web data led to better performance. Even when labeled data was completely unavailable, as in screen similarity, models trained only on web data could often be directly applied to mobile app screens. Our results suggest that the size of current UI datasets may be a limiting factor, since model performance increases consistently when trained on larger splits of data. Our observations and analysis of WebUI's composition showed that web pages can differ from mobile app screens in terms of complexity (*i.e.,* average number of on-screen elements) and element types. However, the performance improvements from our machine learning experiments suggest that web and mobile UIs are similar enough to transfer some types of semantics between them.

We currently only explored three examples, although we believe that other UI modeling works [64, 294, 326] can also benefit from similar approaches. We did not evaluate all possible applications of WebUI in our paper, due to time and cost constraints. However, the three experiments we conducted cover all possibilities of source and target domain labels (4.1), so similar transfer learning techniques are likely to apply. Future work that builds upon WebUI can conduct more detailed evaluations of other downstream tasks.

One specific area that we believe is promising for future work is automated design verification [230], which could benefit from a large volume of web pages containing paired visual and stylistic information. Our highly automated data collection process also allows WebUI to be more easily updated in the future by re-visiting the same list of URLs. An updated version of the dataset could also facilitate longitudinal analysis of the design [82] and accessibility [105] of web UIs. Nevertheless, WebUI is currently unlikely to support other types of modeling, such as user interaction mining [79, 80], that require realistic interaction traces, since our crawling strategy was largely based on random link traversal.

4.5.2 Improved Automated Crawling

Our crawler was unable to access much of the "deep web" (*i.e.,* large part of the web that cannot be indexed), and thus our dataset contains few, if any, web pages that are not publicly accessible or protected by authentication flows. It also did not attempt to interact with all elements on a web page and conducted a very limited exploration of any JavaScript-enabled functionality that might have been present. Trends in web and app development, such as the creation of Progressive Web Apps (PWAs), suggest that this type of functionality will become more common, and traditional link-based traversal may become less effective at exploring UI states.

To improve automated crawling and data collection, our crawler could benefit from a semantic understanding of web pages. For example, it could detect page functionality to explore states that require human input and either execute automated routines (*e.g.* detecting login fields) or employ crowdsourcing [80] to allow it to proceed in more complex scenarios. Our currently trained models could augment or improve this process by identifying tasks associated with web pages (*e.g.,* screen classification) or by augmenting potentially noisy labels provided by the automatically generated accessibility tree. In turn, the crawler could explore more of the web, leading to higher quality and more diverse data. If repeated iteratively, this process would constitute a form of Never-Ending Learning [229], a machine learning paradigm where models learn continuously over long periods of time. Instead of learning from a fixed dataset, models could constantly improve itself by encountering new content and designs, both of which are important due to the dynamic nature of UIs.

4.5.3 Generalized UI Understanding

Our experiments show that incorporating web data is most effective for improving visual UI modeling in transfer learning settings where a limited amount of target labels are available for fine-tuning. A logical next step is to obtain similar benefits without any additional labeled data.

To this end, we identified several strategies for improving generalization. First, unlike existing UI datasets that contain examples from one device type, we intentionally simulated multiple viewports and devices during data collection. The decomposition of one-hot labels (where each element type is assigned exactly one type) into combinations of multi-hot tags (each element can be assigned multiple labels) may also be useful, since it avoids the problem of platform-specific element types. Figure 4.6 demonstrates the zero-shot transfer capabilities of models trained only on web data by successfully detecting and classifying elements on Android app screens. While the label sets of web and Android data do not directly overlap, the web model outputs reasonable analogs (*e.g.,* Text, link) for Android widgets (*e.g.,* Text Button). Finally, our screen similarity model shows how *unsupervised domain adaptation* can improve the transferrability of learned features across domains through an explicit machine learning objective.

A long-term goal of our automated data collection and modeling efforts is achieving a more generalized understanding of UIs — a single model that could be used to predict semantics for any UI. This is challenging due to differing design guidelines and paradigms, but it could ultimately lead to a better understanding of how to solve UI problems across platforms.

4.6 Conclusion

In this paper, we introduced WebUI, a dataset of approximately 400,000 web pages paired with visual, semantic, and style information to support visual UI modeling. Unlike most existing datasets for UI research that depend on costly and time-consuming human exploration and annotation, WebUI was collected with a web crawler that uses existing metadata, such as the accessibility tree and computed styles, as noisy labels for visual prediction. Our highly automated process allowed us to collect an order of magnitude more UIs than other publicly released datasets and often associates more information (*e.g.,* clickability, responsiveness) with each example. We demonstrated the utility of our dataset by incorporating it into three visual UI modeling tasks in the mobile domain: *(i)* element detection, *(ii)* screen classification, and *(iii)* screen similarity. In cases where a small amount of labeled mobile data exists, incorporating web data led to increased performance, and in cases without any labeled mobile data, we found that models trained on web pages could often generalize to mobile app screens. In summary, our work shows that the web constitutes a large source of data that can more sustainably be crawled and mined for supporting visual UI research and modeling.

4.7 Additional Dataset Samples

We provide additional samples from the WebUI (Figure 4.10) to supplement the example in the paper (Figure 4.2). Our example gallery shows several different types of websites, including login, landing, product, portfolio, and informational pages. Each website is captured using different simulated devices, which shows, among other things, how content responds to screen size. We also computed the percentile-rank of each web page's class distribution.

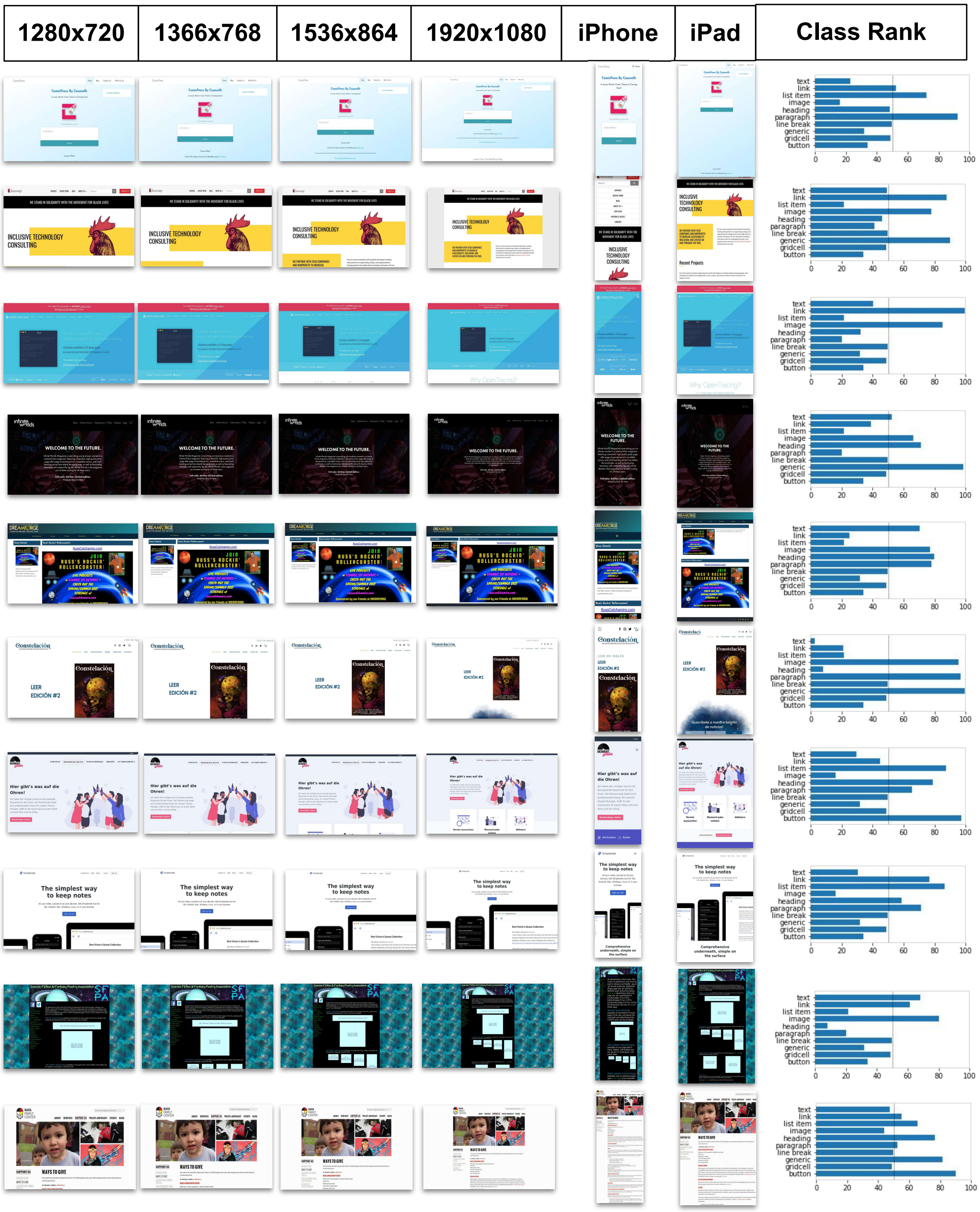


Figure 4.10: Samples from WebUI accessed with different simulated devices. For each screen, we compute its element type distribution (normalized to 1). Then, we computed the percentile-rank of the top 10 classes with respect to the entire dataset. For example, the bottom row's `button` class has a percentile-rank of 90, meaning the web page's relative frequency of is greater than 90% of others in the dataset.

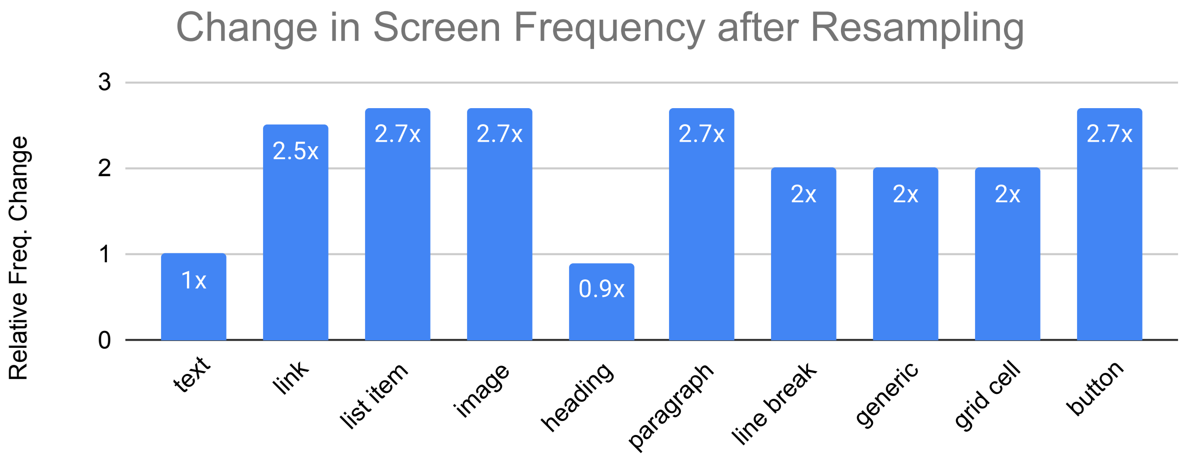


Figure 4.11: We calculated the change in frequency (expressed as a ratio) of screens containing at least one of each element type after resampling. For example, the number of screens containing at least one image element is 2.7x more than in the randomly sampled set.

4.8 Class Imbalance Analysis

This section describes analysis of class imbalance of WebUI and its effect on transfer learning applications. Similar to other UI datasets[347], WebUI exhibits an imbalance of UI element classes, where some types of elements (*e.g.,* text) appear much more frequently than others (*e.g.,* images). Several aspects of WebUI (*e.g.,* finer-grain text segmentation, multi-hot labels, and prevalence of documents on the web) also contributed to class imbalance.

We used a frequency-based resampling method to generate the Web7k-Resampled, which resulted in more examples of infrequent element types. Our technique assigned weights to samples to increase the representation of UIs containing rare or infrequent element types, and we resampled based on the 10 element types shown in Figure 4.3. Algorithm 1 provides an overview of our resampling technique. Note that unlike some class-balancing algorithms (*e.g.,* SMOTE [60]), our technique does not generate additional synthetic samples and does not include the same screen more than once.

Web7k-Resampled contains proportionally more examples of many infrequent classes (Figure 4.3). Figure 4.11 shows the proportional increase in screens containing each element type. Figure 4.12 shows the proportional increase in the total number of elements for each type.

The results from our performance evaluations in the main paper suggest that this resampled split leads to improvements for each of our three tasks when compared to a randomly sampled subset of the same size. Notably, the element detector model resampled 7k split outperformed the one trained on 70k random split, which suggests that element balancing was particularly useful for tasks where elements types are directly predicted. Tests with other two tasks (screen classification and screen similarity) also led to improvements for the resampled models; however, the gains were more modest. The improvements could be because the element distribution in the resampled split is closer to that of the target data. In addition, we provide a deeper analysis of the Element Detection class, which is most likely to be affected by element type imbalance.

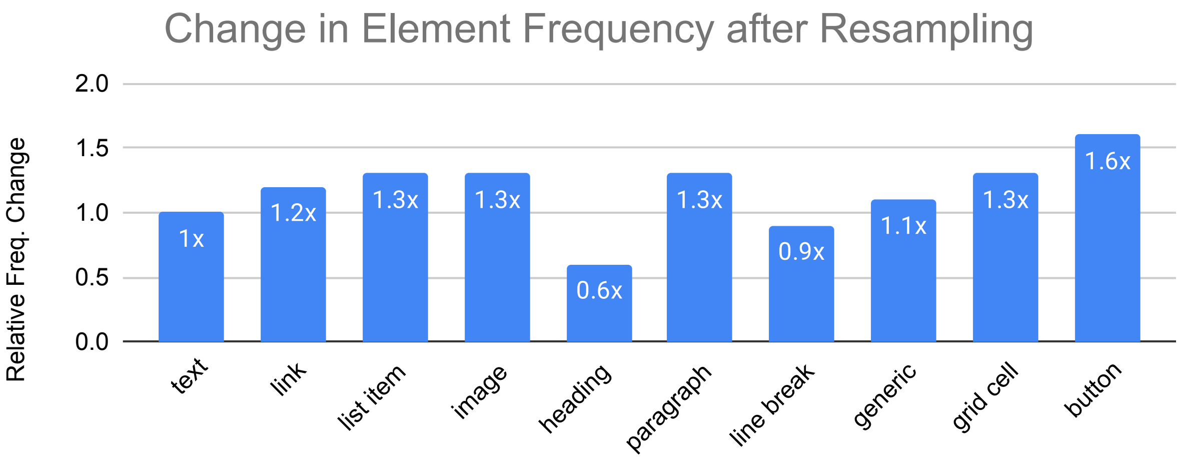


Figure 4.12: We calculated the change in frequency (expressed as a ratio) of total number of elements after resampling. For example, the average screen in the resampled split contains 1.3x more images. Note that is possible for most element classes to increase in frequency (while not having other classes experience a proportional decrease) because element classes are not mutually exclusive, and the resampled split contains more elements that are assigned multiple tags.

Table 4.5: Average Precision (AP) of each element class (excluding the "Other" class) for the Element Detection task.

Element Type	SSD (Random)	FCOS (Random)	FCOS (Web7k)	FCOS (Web7k-Re.)	FCOS (Web70k)	FCOS (Web350k)
Background Image	0.85	0.88	0.86	0.91	0.85	0.93
Checked View	0.06	0.28	0.31	0.34	0.32	0.38
Icon	0.72	0.73	0.75	0.75	0.75	0.77
Input Field	0.22	0.59	0.7	0.60	0.72	0.69
Image	0.73	0.8	0.77	0.82	0.78	0.82
Text	0.66	0.83	0.89	0.84	0.9	0.85
Text Button	0.57	0.9	0.94	0.94	0.95	0.94
Page Indicator	0.83	0.76	0.83	0.76	0.79	0.8
Pop-Up Window	0.85	0.83	0.8	0.85	0.78	0.83
Sliding Menu	0.95	0.98	0.96	0.98	0.96	0.97
Switch	0.97	0.93	0.86	0.97	0.91	0.94
mAP	0.67	0.77	0.79	0.80	0.79	0.81

Algorithm 1: Pseudo-code for the frequency-based resampling algorithm used to generate the Web7k-Resampled split.

```
function SampleSplit ($N, C, S$);
Input  : Number of samples to choose $N$, list of element classes $C$, and list of samples
         $S$
Output: Resampled subset of $S$
/* Vector containing total frequencies for $c \in C$                          */
$f_C \leftarrow$ total # of elements in $S$ for each class
/* Matrix where rows are $s \in S$ and columns are normalized
   frequency of $c \in C$ for $s$                                              */
$f_S \leftarrow$ frequency of classes $c \in C$ (columns) for $s \in S$ (rows)
/* Assign sampling weights to $c \in C$ inversely proportional
   to frequency                                                                */
$w_C \leftarrow [\frac{1}{f_C[c]}$ — $c \in C]$
samples $\leftarrow$ []
/* Repeat until desired split size is reached                                  */
while len(samples) $< N$ do
    $c_s \leftarrow \mathbf{Sample}(C, w_C)$
    $w_s \leftarrow [f_S[s, c_s]$ — $s \in S]$
    sample $\leftarrow \mathbf{SampleWithoutReplace}(S, w_s)$
    add sample to samples
end
return samples
```

Table 4.5 shows that the Web7k-resampled split has higher AP for classes like ”Text Button” and ”Image”, which had increased representation after resampling.

5

Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots

5.1 Preamble

Content in this chapter is based on a conference paper that was published at UIST 2021 [326].

In the previous chapter, I introduced several models for predicting semantic information about UIs, such as the *i)* the location of UI elements and *ii)* a screen categorization. While these models allow systems to make basic inferences about existing apps, they fall short of the descriptive resolution required by assistive technology. Assistive technology often relies on a hierarchical representation of the UI known as the *accessibility tree*. The accessibility tree is a model of the information and widgets are presented to the user (*i.e.,* presentation model). In this chapter, I present a system that reverse-engineers this semantic information from a single UI screenshot.

Automated understanding of user interfaces (UIs) from their pixels can improve accessibility, enable task automation, and facilitate interface design without relying on developers to comprehensively provide metadata. A first step is to infer what UI elements exist on a screen, but current approaches are limited in how they infer how those elements are semantically grouped into structured interface definitions. In this paper, we motivate the problem of *screen parsing*, the task of predicting UI elements and their relationships from a screenshot. We describe our implementation of screen parsing and provide an effective training procedure that optimizes its performance. In an evaluation comparing the accuracy of the generated output, we find that our implementation significantly outperforms current systems (up to 23%). Finally, we show three example applications that are facilitated by screen parsing: *(i)* UI similarity search, *(ii)* accessibility enhancement, and *(iii)* code generation from UI screenshots.

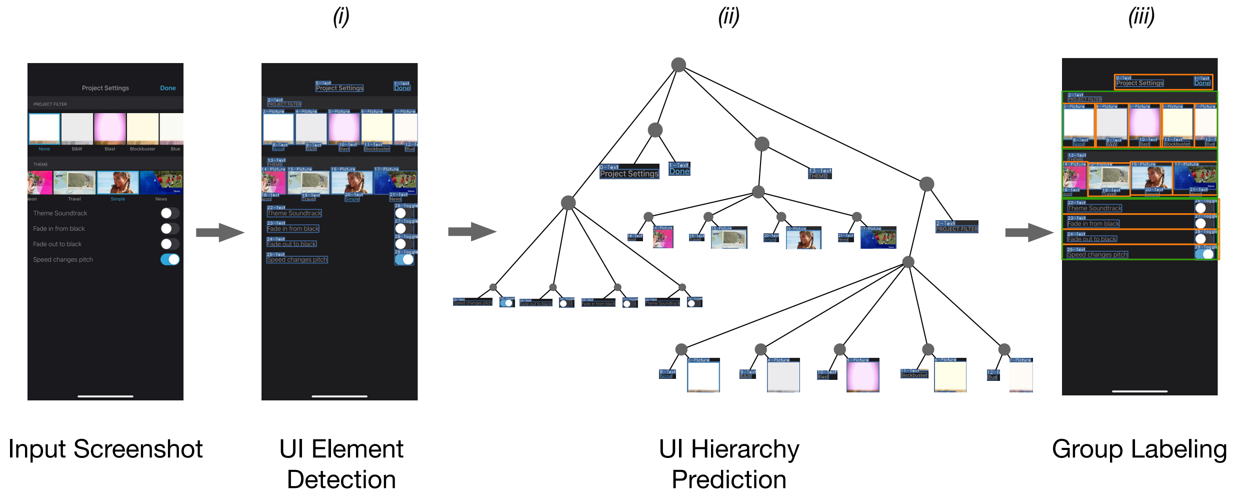


Figure 5.1: An overview of our implementation of *screen parsing*. To infer the structure of an app screen, our system *(i)* detects the location and type of UI elements from a screenshot, *(ii)* predicts a graph structure that describes the relationships between UI elements, and *(iii)* classifies groups of UI elements.

5.2 Introduction

User interfaces are, unsurprisingly, designed for consumption by human beings, and it can be difficult for automated systems to understand what functionality is present in a user interface, how the different components of the interface work together, and how it can be operated to accomplish some goal. This is particularly true if the automated system does not have access to any meta-data about the user interface, such as view hierarchies or accessibility tags, or if this information is missing or incompletely defined, as is often the case. Automated user interface understanding systems could offer many benefits. For example, a screen reader (*e.g.,* VoiceOver and TalkBack) could facilitate access to user interfaces for blind and visually impaired users when the underlying app does not provide appropriate meta-data [347], and task automation agents (*e.g.,* Siri Shortcuts and IFTTT) could allow users to automate repetitive or complex tasks with their devices more efficiently. These benefits are gated on how well these systems can understand and interact with the underlying applications. Many of today's systems rely on the availability of UI meta-data and fail when this information is unavailable. To overcome this recent efforts have focused on predicting the presence of an app's on-screen elements solely from its visual appearance.

Structure is a core property of UIs that is reflected both in how they are constructed and how they are used. However, many current approaches to visual modeling of UIs ignore or fail to centralize this aspect. In this paper, we present a new approach called *screen parsing*, which applies techniques used in NLP for natural language parsing to produce machine-learned models that predict the UI hierarchy of an app from its screenshot. Our approach involves *(i)* a Faster-RCNN model for detecting the set of elements on a screen, *(ii)* a stack-based transition parser model for predicting the hierarchy of how those elements relate to each other, and *(iii)* a Deep Averaging Network model that classifies element groupings. We describe the details and training procedure of our implementation of screen parsing, and conduct an evaluation in which we compare the performance of our system against baseline approaches. Using a set of 5 metrics, we show that our implementation performs up to 23% better than baseline systems depending on the performance metric used. Finally, we show three example applications enabled by our implementation of screen parsing.

More broadly, we believe that systems can benefit from perceiving UI screens as humans do – not as a set of elements, but as a coordinated and organized presentation of content. Structural understanding is an important step that can help systems reason about relationships between interaction controls and content. Our model implementation is trained to predict one type of relation (links in the view hierarchy), but we believe screen parsing and our modeling approach can be extended to others as well (*e.g.,* navigation order).

To summarize, this paper makes the following contributions:

- A problem definition of *screen parsing* which is useful for a wide range of UI modeling applications.
- A description of our implementation of screen parsing and its training procedure.
- A comprehensive evaluation of our implementation with baseline comparison.
- Three implemented examples of how our model can be used to facilitate downstream applications such as *(i)* UI similarity, *(ii)* accessibility metadata generation, and *(iii)* code generation.

5.3 Screen Parsing

5.3.1 Problem Formulation

We define the problem of *screen parsing*, which we use to refer to the prediction of structured UI models from visual information. As a review, we use *UI models* to refer to high-level abstractions of UI semantics *e.g.,* logic, presentation, and associated tasks [125]. A *screen parsing* model takes a UI screenshot as input and produces a graph representation of a model as output. The connections in the graph can be used to express a variety of semantic and syntactic concepts. For example, one might use an edge to represent interaction flow (*e.g.,* the "Username" text field should be filled out before tapping on the "Login" button).

In this paper, we focus on generating an app's UI hierarchy (*i.e.,* presentation model) which is a specification of how UI elements are grouped and rendered on the screen [257]. Figure 5.2 shows an example of a screen and corresponding UI hierarchy graph. The properties of UI hierarchies introduces some constraints on the types of valid outputs.

- *Complete* – the output is a single tree that spans all of the detected UI elements.
- *Grounded* – Nodes in the output reference specific UI elements in the screen.
- *Abstractive* – The output can group elements together (potentially more than once) to form higher-level structures.

Moreover, all UI hierarchies can be described as *directed trees*, which we constrain our system to generating.

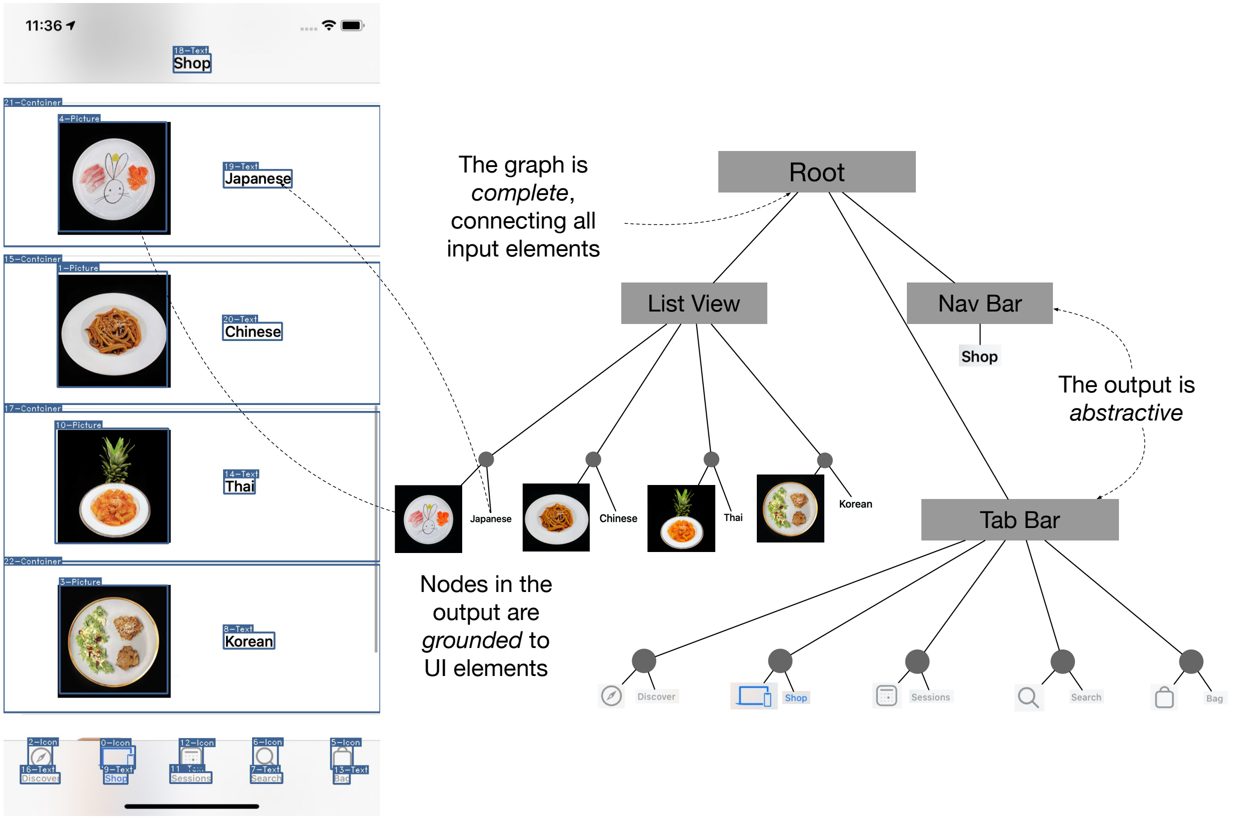


Figure 5.2: We show an example of an input screen (Left) and the corresponding *screen parse* (Right). The graph contains all of the visible elements on the screen (the output is *complete*), groups them together to form higher-level structures (*abstractive*), and nodes can be used to reference UI elements (the output is *grounded*).

Table 5.1: This table shows the requirements of several downstream applications and support for them among our implementation and related approaches. Screen parsing's problem formulation allows it to be applied more widely.

Requirements	Complete	Grounded	Abstractive
Applications			
Structural Similarity	N	N	Y
Screen Reader	N	Y	N
Code Generation	Y	N	N
Approaches			
Scene Graph	Y	Y	N
GUI Skeleton	N	N	Y
Heuristics	Y	Y	N
Our Implementation	Y	Y	Y

5.3.2 Comparison to Related Problems

Screen parsing is closely related to and, in many ways, motivated by other problems in the UI modeling and computer vision. Specifically, we select three similar tasks for comparison: *(i)* UI Element Detection, *(ii)* GUI Skeleton Generation, and *(iii)* Scene Graph Generation. All of these approaches generate semantic output from a visual representation of a screen (*i.e.,* screenshot). However, there are important differences that make *screen parsing* applicable to a wider range of down-stream applications (Table 5.1).

UI Element Detection

UI Element Detection is a specific application of object detection, which extract a *set* of class-labeled bounding boxes from an image. When trained and applied to UI screens, the prediction output corresponds to the set of UI elements on the screen, which is useful on its own or as a "first-pass" step for further processing. The main difference from *screen parsing* is that UI Element results in a flat structure, which prevents it from representing relationships between elements. Heuristics can be applied to detect and group elements; however there is no guarantee that all elements will be connected.

GUI Skeleton Generation

The GUI Skeleton is an artifact produced by the UI2Code system that describes the types of widgets in a screen and their hierarchical structure [61]. Similar to our model implementation, UI2Code is trained to produce trees processed from view hierarchies.

It is important to note that an app's GUI Skeleton is different its UI hierarchy (the target output of our model). Namely, it doesn't support we what refer to as *element grounding*, the ability to match items in its output to its input. For example, an app's GUI Skeleton might indicate that the screen contains a list container with three buttons, but it is unable to indicate which three buttons (on a screen with many buttons) belong to the list. Thus, the GUI Skeleton cannot be used to support certain applications, such as screen reader navigation.

Scene Graph Generation

Screen Graph Generation (SGG) is a visual scene understanding problem that models the relationships between visible objects using *scene graphs*. Like our model, SGG models are designed to process an input image and generate a graph whose nodes are detected objects in the scene and edges are semantic relationships between those objects.

Scene graphs are often constructed to describe real-world visual scenes [167]. Unlike UIs, which are typically constructed using nested views stemming from a single root node, visual scenes can contain multiple entities, represented as independent sub-graphs. We purposefully constrained our model to produce a single connected tree to reflect this property of UIs.

Most edges in a scene graph correspond to direct relationships between detected objects, and SGG models often consider *pairwise* relationships rather than *hierarchical* ones. Because of this, a strong and frequently-used baseline for SGG is computing the prior probabilities of relationships between object classes (ignoring position) on the training set [343]. Edges between leaf nodes are relatively rare among UI hierarchies, as most elements are indirectly joined by container elements.

5.4 Implementation

Our implementation of screen parsing uses separate models to *(i)* detect elements from a screenshot, *(ii)* group them together in a graph structure, and *(iii)* predict labels for the element groups.

5.4.1 UI Element Detection

We used a standard object detection model to extract the set of UI elements in a screen and their parameters. Specifically, we trained a Faster-RCNN [268] model with a ResNet-50 [134] backbone on our UI screen dataset. Before feeding an image to the element detection model, we resized images to 256x256 and normalized each input channel to have a mean of 0 and standard deviation of 0.5. We first run our detection model on an input screenshot and keep all detections that have a confidence of at least 0.7. We then apply non-max suppression to remove overlapping detections with lower confidence (IoU threshold of 0.5).

5.4.2 UI Hierarchy Prediction

After a set of detections is obtained from the Element Detection model, the next step is to predict their hierarchical relationship. A natural way of representing this is using a graph structure, where elements are linked to one another with parent-child relationships. Intuitively, the problem can be thought of as generating a complete graph (*i.e.,* the UI hierarchy) given the leaf nodes (*i.e.,* visible elements). We draw inspiration from the NLP literature on text parsing, where such graph structures are often used to define relationships between words in a sentence. Specifically, we

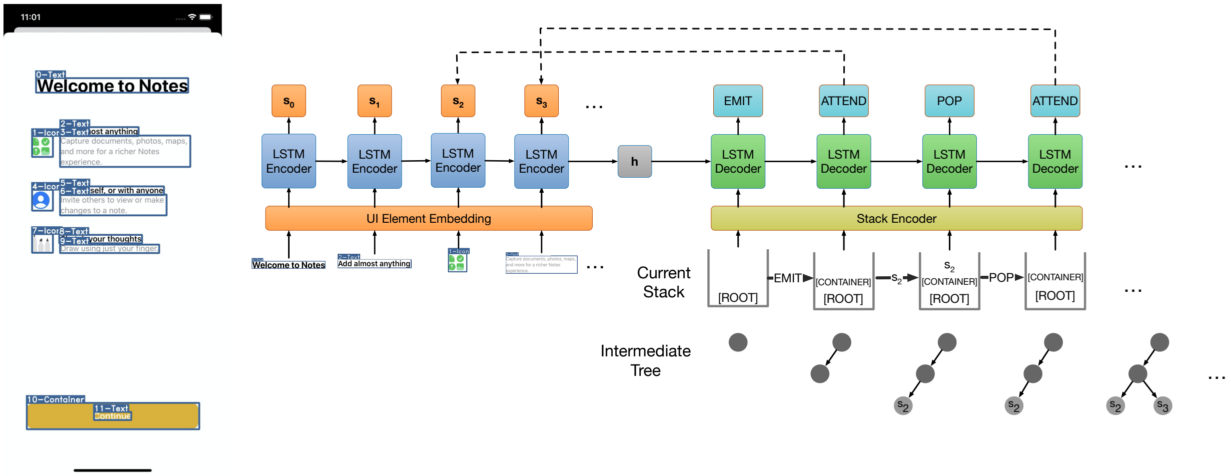


Figure 5.3: Our UI Hierarchy prediction model is a stack-based transition parser. A Bi-directional LSTM encoder is fed a set of embedded UI elements and query tokens. The final hidden state is used to initialize a LSTM decoder network. The decoder produces a sequence of actions that describe the UI hierarchy using a continuously updated state (stack, buffer, and visited set).

build a top-down transition-based parser [218], which is able to construct any UI hierarchy[1], and offers fast and efficient decoding.

Like other transition-based parsers, our model incrementally produces a graph structure through a sequence of actions, and is most closely related to the approach detailed in similar dependency parsers used in NLP [218]. Our model uses three data structures to perform parsing: the input buffer (l) that holds the set of visible UI elements, the stack (σ) that allows the model to traverse the graph, and the set of visited nodes (α). The actions that we support are:

- *Arc* – A directed edge is created between the node on top of σ (parent) and the node in $l - \alpha$ with the highest attention score (child). The child is pushed onto σ and added to α.
- *Emit* – An intermediate node (represented as a zero-vector) is created and pushed onto σ.
- *Pop* – σ is popped (*i.e.,* the top element is removed).

Figure 5.3 provides an example of how these actions are used to parse a screen.

Model Architecture

Our model architecture (Figure 5.3) consists of a LSTM-based encoder and decoder. Our chosen encoder model, the LSTM [138], is a type of recurrent neural network effective at encoding long sequences. LSTMs are designed with special gated memory cells that enable it to perform computations useful for our task, such as counting and recognizing hierarchy [132]. The input of the model is the list of UI elements in a screen, sorted using y-position as the primary key and x-position as the secondary key. Each element represented as the concatenation of its position

[1] Some parsing algorithms are designed to handle only a subset of parse trees known as *projective trees*, which makes them difficult to apply to view hierarchies.

and a one-hot class vector for the UI element type (*e.g.,* Text, Slider, Picture, etc...). The final hidden state is used as the initial state of the decoder.

Decoding

At every decoding timestep, the LSTM is fed the *(i)* last hidden state (h_t) and *(ii)* the element at the top of σ. The LSTM returns *(i)* an output vector (o_{t+1}) and *(ii)* an updated hidden state (h_{t+1}). The output o_{t+1} is fed through a linear layer that produces the logits for the emit and pop actions. The output o_{t+1} is also used to compute the scaled dot-product attention between all of the encoded UI elements $\{s_0, s_1, ..., s_N\}$. Finally, an action vector is constructed by concatenating the emit (u_e^i) and pop (u_p^i) activations with the attention scores.

$$u_j^i = \frac{{e_j}^T h_i}{\sqrt{n}} \tag{5.1}$$

$$p(a_i|a_0, a_1, ..., a_{i-1}, P) = softmax(concat(u_e^i, u_p^i, u^i)) \tag{5.2}$$

n is the size of the hidden state, P represents the input and $a_0, a_1, ...a_n$ represent the previously selected actions. This process is repeated until all leaf nodes are added to α, which guarantees that the generated graph is complete. Finally, as a heuristic to prevent repeated *Emit* and *Pop* actions, we set the probability of the *Emit* action to 0 if the last 10 actions does not contain an *Attend.*

The output of the model is additionally smoothed to remove extraneous intermediate nodes.

5.4.3 Group Labeling

Table 5.2: Table of group labels considered for each dataset, along with number of occurrences.

AMP	RICO
Tab Bar Button (63170)	List Item (56186)
Table (23693)	Toolbar (29068)
Tab Bar (19602)	Card (6091)
Collection (19420)	Drawer (5756)
Button (9779)	Multi-Tab (3189)
Segmented Control (2988)	Bottom Navigation (236)

To label the intermediate nodes in a tree, we train a separate classifier. We first inspect each dataset to determine the most common labels assigned to "containers" and select 7 classes (including an "Other" class) based on frequency and relevance to our task (Table 5.2).

Our Group Labeling classifier is based off the Deep Averaging Network (DAN) architecture used for sentence classification [152]. To classify a given node, we retrieve a list of all of its descendant elements. Each element in the list is embedded using using a feed-forward layer, and all of the embeddings are pooled using the sum operation. The pooled representation is fed into a MLP that predicts its label. Because some containers appear much more frequently than others, we use a weighted loss function for training (class-weighted cross entropy), and the F1-macro metric to measure validation and test performance. Our best group labeling models achieved F1-macro scores of 0.61 and 0.76, on AMP and RICO (our two training datasets).

This approach to classifying element groups is a simple one that does not model the joint probability of multiple element groups (*e.g.,* the probability of one group's label conditioned on another's). We will improve this aspect of our system in future work.

5.5 Training

In this section, we primarily describe the training procedure for our system's primary component – the UI Hierarchy model. We first describe how we extracted and processed a dataset for this purpose. Then, we describe an effective approach for training parsing models that is especially relevant to UI Hierarchy modeling.

5.5.1 Datasets

We trained our models on two mobile UI datasets: *(i)* **AMP**, an internal dataset of 130,000 iOS screens, and *(ii)* **RICO**, a publicly available dataset of 80,000 Android screens [80]. Each dataset contains screenshots, annotated screens, and their view hierarchies. Both datasets collected by crowdworkers who installed and explored popular apps across 20+ categories (in some cases excluding certain ones such as games, AR, and multimedia) on the iOS and Android app stores. More information is available in the original papers [80, 347]. Before training, three splits are created for each dataset: training (70%), validation (15%), and testing (15%). When training our system, we only train on screens with less than 64 elements (to make training more efficient), but we do not apply this constraint to our test set.

Node Correspondence

The first step is to match up visible elements with a corresponding node in the view hierarchy. We ran our trained UI Element detector on screenshots, which produced a list of detections above a confidence threshold (0.7). We employed a best-cost matching algorithm [169] to compute the best match between the set of element detections and the set of bounding boxes found in the view hierarchy. The matching score between two bounding boxes are defined as the intersection-over-union (IoU) score, and pairs with low scores (IoU ¡ 0.5) are ignored.

Extracting Hierarchical Information

We found that many of the screens in our dataset had missing or mostly incomplete view hierarchies (*i.e.,* most of the visible elements did not have a corresponding element in the view hierarchy). For example, in the **AMP** dataset, we found that around 40,000 screens had view hierarchies that were suitable for ground truths. To train and evaluate our model on a higher-quality subset, filtered both datasets. The AMP dataset was filtered by selecting screens where at least 80% of annotated nodes had a corresponding element in the view hierarchy. The RICO dataset was filtered using scores from the node correspondence step – only screens where the average match score was greater than 0.8.

Graph Smoothing

Because the view hierarchy is an artifact of the UI rendering system, it contains some irrelevant nodes and edges that represent class inheritance and singleton containers. We preprocessed view hierarchy graphs using a smoothing algorithm that removed nodes which *(i)* only had one child and *(ii)* did not correspond to a visible element.

5.5.2 Training Algorithm

A standard approach to training transition-based parsers is defining an "oracle" function that produces a sequence of actions for every view hierarchy. An example of an oracle function for graph-structured data is running a depth-first search and recording the order nodes were entered and exited (Figure 5.4). We compared two different approaches to oracle training for our element grouping model.

The first approach we compared is the *static oracle*, which is a simple and common implementation that traverses the graph deterministically (*i.e.,* produces exactly one sequence of actions for every graph). For screen parsing, this requires defining an ordering function that sets a deterministic order by which children are processed (*e.g.,* children are ordered top-down, left-to-right). During training, the parser is trained to maximize the likelihood of the static oracle's "gold" action at every timestep.

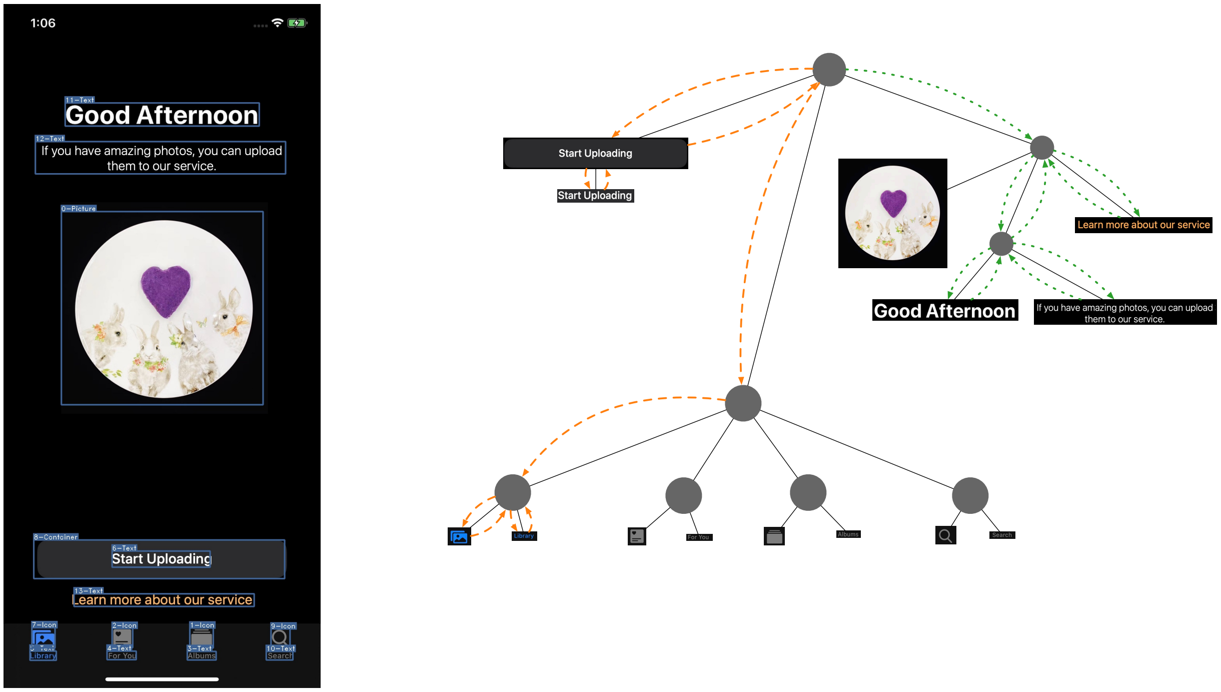
The second approach is a *dynamic oracle*, which provides a *set* of optimal actions at every state for the model to learn instead of a single action. During training, if the model's top-choice action is optimal then it is executed, and otherwise an optimal action from the oracle's output is randomly selected and executed. While other options are available [34, 121], we found that training to maximize the average likelihood of the set of optimal actions [221] led to the best results:

$$p(\mathbf{z_g}|\mathbf{p_t}) = \frac{1}{|\mathbf{z_g}|} \sum_{z_{g_i} \in \mathbf{z_g}} p(z_{g_i}|\mathbf{p_t}) \tag{5.3}$$

5.6 Evaluation

We compared our final system (Screen Parser Dynamic) to *(i)* a baseline system [347] and *(ii)* a baseline training procedure [218], and we show that our implementation significantly improves performance.

Screen Recognition is a heuristic-based system used to generate accessibility metadata from pixel data, and it is similar to heuristic-based approaches employed by other UI reverse-engineering work [238]. Similar to our system, Screen Recognition first runs an object detection model on a screenshot, which returns a set of element detections. These detections are then processed using a set of manually-defined heuristics that check for features such as nesting, text grouping, tab grouping, and picture subtitles.



Static Oracle Sequences

8, 6, POP, POP, EMIT, EMIT, 7, POP, 5, POP, …

Dynamic Oracle Sequences

8, 6, POP, POP, EMIT, EMIT, 7, POP, 5, POP, …
8, 6, POP, POP, EMIT, EMIT, EMIT, 11, POP, 12, ..
8, 6, POP, POP, EMIT, EMIT, 11, POP, 10, POP …
Explores left subtree first

EMIT, EMIT, 0, POP, 13, POP, EMIT, 1, POP, 3, POP …
EMIT, EMIT, 13, POP, EMIT, 11, POP, 12, POP, POP, …
EMIT, EMIT, EMIT, 11, POP, 12, POP, POP, 2, POP, …
EMIT, EMIT, EMIT, 12, POP, 11, POP, 13, POP, 0, …
Explores right subtree first

EMIT, EMIT, 7, POP, 5, POP, POP, EMIT, 2, POP, 4 …
EMIT, EMIT, 1, POP, 3, POP, POP, EMIT, 2, POP, 4 …
EMIT, EMIT, 2, POP, 4, POP, POP, EMIT, 5, POP, 7, …
Explores center subtree first

…

Figure 5.4: We explored two oracle-based training procedures for UI hierarchy prediction. For a given ground-truth UI hierarchy (top), a static oracle (left) only produces one sequence of optimal actions, while a dynamic oracle (right) produces all optimal sequences.

We also used a baseline training procedure to train our system (Screen Parser Static), and we show that our chosen approach significantly outperforms standard training methods for NLP parsers.

To summarize, we compared the following systems in our evaluation:

- *RCNN Oracle* - This is not a system; it represents the best possible matching between the RCNN detections and the ground truth hierarchy. This gives a rough bound for the best-case parsing performance given the accuracy of the UI element detector.
- *Screen Recognition* - The complete Screen Recognition with its original UI element detector and heuristics.
- *Screen Recognition + RCNN* - The Screen Recognition heuristics run on the output of our RCNN-based UI element detector. When run on the RICO dataset, we used an RCNN model trained on the RICO dataset and mapped the the labels from the RICO label set to the AMP equivalent (the heuristics were designed for AMP).
- *Screen Parser Static* - The Screen Parser system where the UI Hierarchy model is trained using a static oracle (standard training procedure).
- *Screen Parser Dynamic* - The Screen Parser system where the UI Hierarchy model is trained using a dynamic oracle (improved training procedure).

5.6.1 Performance Metrics

To compare prediction outputs to ground truth view hierarchies, we first used our node correspondence algorithm (Section 5.5.1) to label the nodes in each graph with corresponding identifiers. Container nodes are matched using a similar method, where the score is the IoU of their descendant nodes.

We computed three types of metrics that measure different performance aspects relevant to down-stream tasks.

Edge-based metrics

Popular approaches to evaluating natural language parsers (*e.g.,* constituency parsing) are based on measuring the number of correctly predicted edges (*e.g.,* constituents) [158]. We decomposed both the ground truth and prediction graph into sets of edges and computed two metrics: *(i)* the overall F1 score and *(ii)* the F1 score for only edges that are attached to leaf nodes. The F1 score of the leaves can be more relevant for some downstream applications which use lower-level element groupings.

The F1 score is bounded from 0 to 1 and a higher score indicates better performance. If the prediction doesn't contain any matched nodes (possibly due to inaccurate element detection), the F1 scores for the overall tree and leaves are set to 0.

Distance-based metrics

While edge-based metrics are simple to compute, they can unfairly penalize some types of mistakes (*e.g.,* correct grouping but wrong parent). Graph Edit Distance (GED) is a measurement of graph similarity that considers the minimum number of “edits” needed to make a graph isomorphic to another.

$$GED(g_1, g_2) = \min_{(e_1,\ldots,e_k)\in\mathcal{P}(g_1,g_2)} \sum_{i=1}^{k} c(e_i) \tag{5.4}$$

$\mathcal{P}(g_1, g_2)$ refers to the set of possible edit paths between g_1 and g_2. We consider GED that allows 4 edit operations all with cost of 1: the insertion and deletion of nodes and edges. Exact computation of GED is computationally expensive (NP-complete), so we use an inexact algorithm that approximates an upper bound of the true distance [103].

Because a lower GED indicates better performance, we set the GED to the number of edges in the ground truth tree if the prediction doesn’t contain any matched nodes.

Group-based metrics

Finally, we considered group-based metrics that target the grouping of elements rather than their structure. This metric is more relevant for some downstream tasks such as screen segmentation that aim to partition the screen.

This metric is computed as the mean of each container’s (*e.g.,* intermediate node) IoU score with the ground truth. Similar to edge-based metrics, the container match (CM) score is bounded between 0 and 1, where a score of 1 indicates that all groups were correctly matched. For trees without any matched modes, we set the score to 0.

5.6.2 Results

Table 5.3: We evaluated screen parsing performance using 4 metrics: F1 score (F1), F1 score of edges with leaf nodes (F1 Leaves), graph edit distance (GED), and container match cost (CM). Higher is better for all metrics except GED. More details are described in the performance metrics section. Note that the RCNN Oracle is not a system – it is the best possible matching between the RCNN detections and the ground truth.

	AMP				RICO			
	F1↑	F1 Leaves↑	GED↓	CM↑	F1↑	F1 Leaves↑	GED↓	CM↑
RCNN Oracle	0.76±0.22	0.75±0.22	16.6±20.9	0.79±0.19	0.89±0.14	0.89±0.14	8.8±15.9	0.93±0.07
Screen Recognition	0.40±0.20	0.52±0.26	23.5±20.7	**0.63±0.23**	0.39±0.19	0.47±0.26	23.8±21.4	0.43±0.19
Screen Recognition + RCNN	0.34±0.19	0.44±0.24	25.5±21.1	0.54±0.21	0.41±0.23	0.44±0.28	17.8±19.7	0.48±0.23
Screen Parser Static	0.53±0.23	0.62±0.22	26.1±24.6	0.59±0.16	0.61±0.27	0.59±0.27	15.2±16.2	0.69±0.24
Screen Parser Dynamic	**0.60±0.23**	**0.67±0.23**	**20.2±20.9**	**0.63±0.16**	**0.66±0.28**	**0.64±0.28**	**13.2±15.5**	**0.74±0.24**

Table 5.3 shows the results of our performance evaluation using our set of metrics. Our results show that our final system, Screen Parser Dynamic outperforms all baselines in all performance metrics. In this section, we provide more detailed comparison with baselines and further analyze factors that impact performance.

Comparison with Screen Recognition

Both Screen Parser models outperform Screen Recognition on both datasets. One reason is that Screen Recognition and most other heuristics-based approach are not *abstractive*, which prevents them from producing "deep" trees. Performance on edges containing leaf nodes (*i.e.,* shallower relations) is generally much better; Compared to overall F1 score, Screen Recognition had a 25% higher F1 Leaves score.

More importantly, Screen Recognition was not designed to produce output similar to app view hierarchies; instead, it was designed to support common groupings required by screen reader navigation. In addition to the set of performance metrics described here, we recommend holistically evaluating systems in downstream tasks.

Effect of Improved Training Procedure

Based on our results, Screen Parser Dynamic performs up to 23% better than Screen Parser Static. Since both static and dynamic versions of our model was trained to maximize the likelihood of the same data, we can conclude that the dynamic oracle training technique is effective in increasing screen parsing performance.

Recall that the main difference between the two training procedures is that the static oracle only produces one sequence of optimal actions (*i.e.,* the canonical action sequence) while the dynamic oracle produces all optimal sequences (Figure 5.4). This is especially relevant for UI hierarchies, where the tree structure can be several levels high, leading to exponentially more possible optimal sequences. This is in contrast to natural language parse trees, which are typically limited by the relatively short length of sentences.

While the canonical action sequence provably correct (*i.e.,* contains all correct element relationships) [121], it leads to *exposure bias* – where the model is biased to perform well only in states it has seen during training. During test-time, the model may choose an action outside of this sequence (either by making an error or choosing another optimal action), which causes the model to perform poorly afterwards.

Effect of UI Element Detection Performance

All systems were fed UI element detections as input, and errors in the upstream model also affected the performance of the hierarchy prediction.

To estimate the upper-bound performance of systems that rely on the RCNN output, we included the *RCNN Oracle* which constructs an output using the best possible between the detector output and the ground truth hierarchy. Even with access to the ground truth, it does not achieve perfect accuracy – possibly a result of missing or inaccurate detections. This suggests that a better object detection model could further improve UI hierarchy prediction.

As an example, we ran Screen Recognition's heuristics on both its default object detector and our RCNN model's output. Compared to our system's RCNN model, Screen Recognition's object detector is optimized for the AMP dataset (*e.g.,* tuned per-class confidence threshold) which results in better performance.

Performance on Complex Screens

Finally, we analyzed the performance of screen parsing system on screens of different complexity. Figure 5.5 shows the overall F1 score for each system run on splits of the test data containing a screens with a specified # of elements. Performance is highest for screens up to 32 elements and degrades following that threshold. One major factor is lower object detection accuracy with smaller objects (screens with more elements tend to have smaller elements), since the performance of the RCNN Oracle also drops past that point. Interestingly, Screen Recognition's performance remains relatively constant, which suggests that many of the local patterns targeted by heuristics are not as affected by screen complexity. We also note that although both Screen Parser systems were only trained on screens with up to 64 elements, they still perform competitively for more complex screens.

Examples of failure cases (some of which result from these factors) are shown in Figure 5.6.

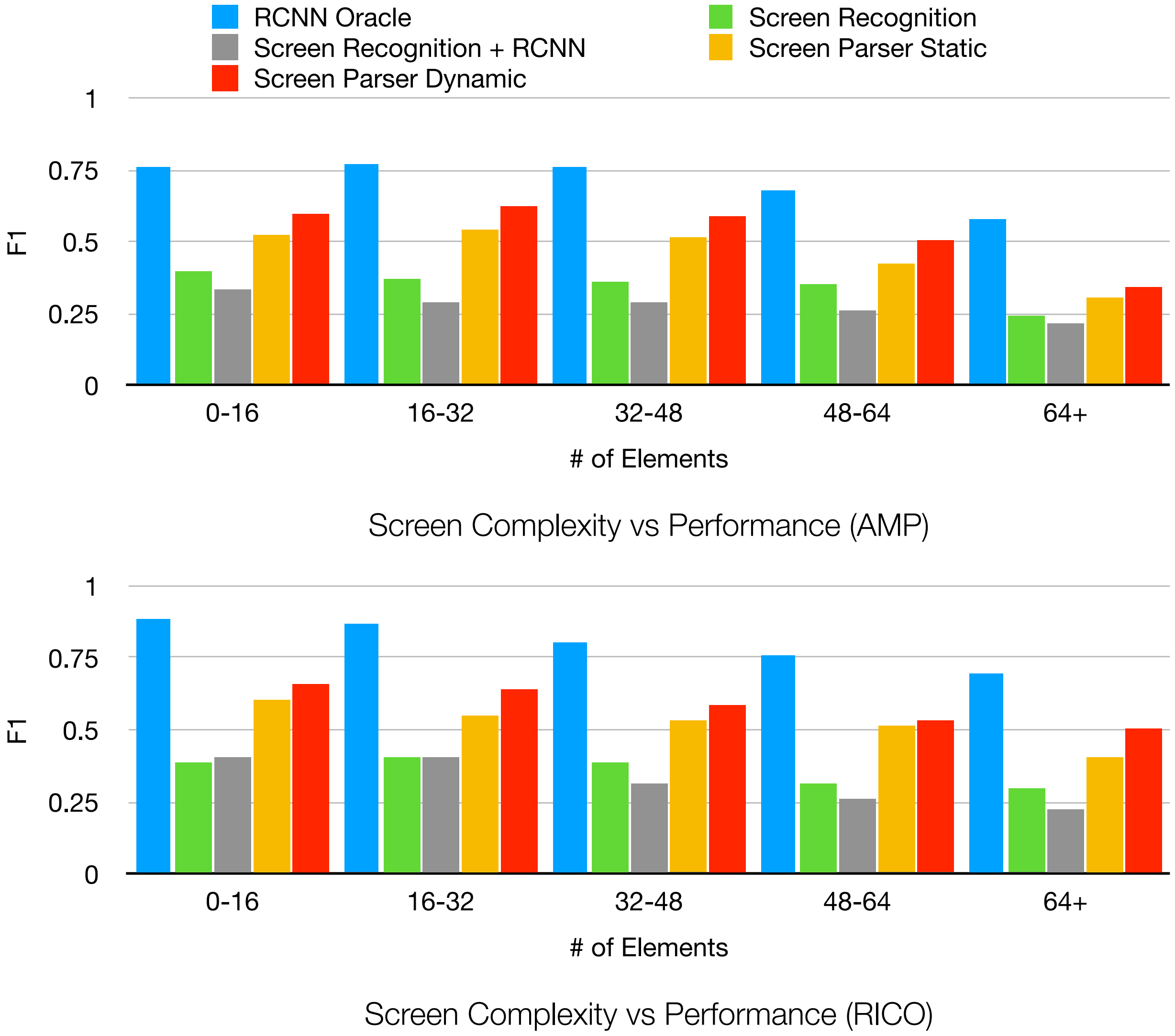
5.7 Example Applications

In this section, we present a suite of example applications implemented using our screen parsing model. These applications show the versatility of our approach and how the UI hierarchy predicted by our model can be used to facilitate many existing tasks.

5.7.1 UI Similarity Search

Many recent efforts in modeling UIs have focused on representing them as fixed-length embedding vectors. These vectors can be trained to encode different properties of UI screens (*e.g.*, layout, content, and style) and support down-stream tasks. For example, a common application of embedding models is measuring screen similarity, which is represented by distance in embedding space. We believe the performance of such models can be improved by incorporating structural information, an important property of UIs.

Our implementation is trained to model the structural relationships between on-screen elements, and we show that its internal representations are useful for this purpose. To generate an embedding of a UI, we feed it into our model and pool the last hidden state of the encoder. This includes information about the position, type, and structure of on-screen elements. Figure 5.7 shows the 2-D projection [224] of randomly-sampled screens embedded using this technique. This set includes several variations of app screens, including *(i)* scaling, *(ii)* language, *(iii)* theme, and *(iv)* dynamic content. Our model is largely invariant to these changes, since their structure is the same, just rendered under different conditions. The properties of our embedding could be useful for some UI understanding applications, such as app crawling and information extraction where screens are characterized more by their semantics than appearance. We provide examples of UIs retrieved by our similarity search application in the appendix to illustrate the types of information our embedding captures.



Screen Complexity vs Performance (AMP)

Screen Complexity vs Performance (RICO)

Figure 5.5: Analysis of each system's performance on screens of varying complexity. Screens with a higher number of elements introduce challenges for both UI element detection (screens with large # of elements generally have smaller and more dense elements) and UI hierarchy prediction.

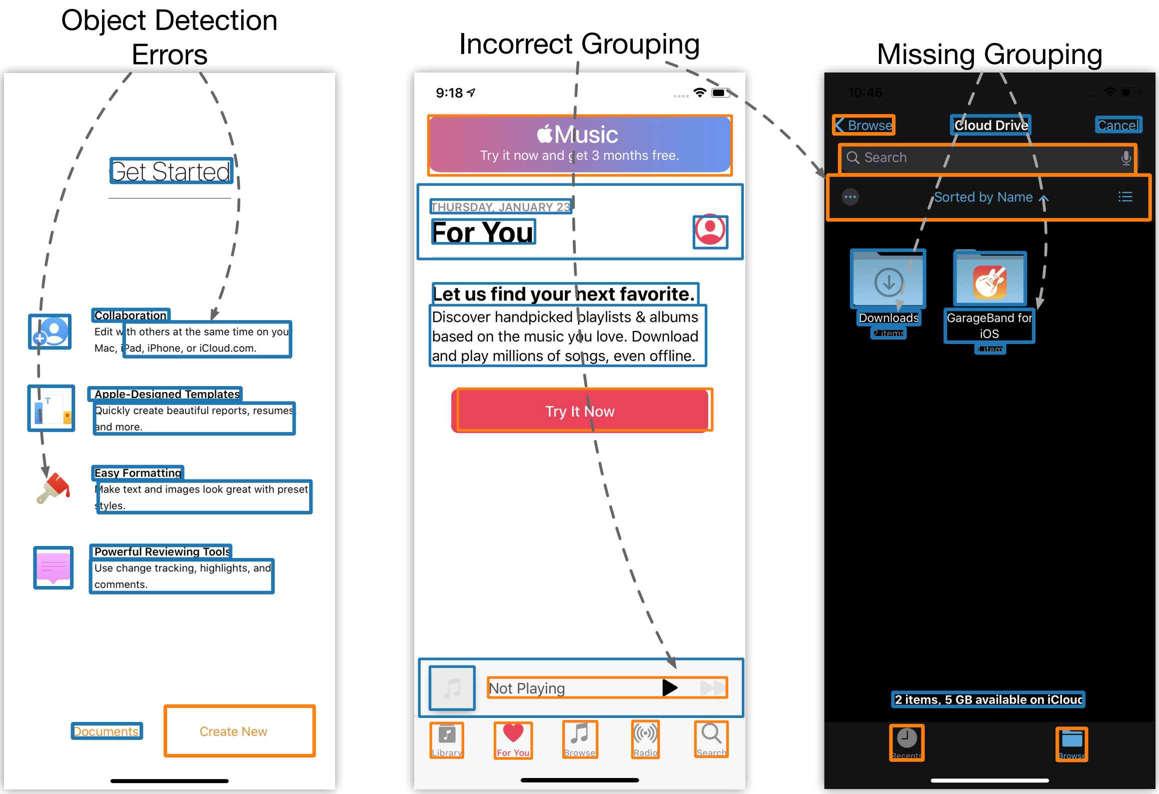
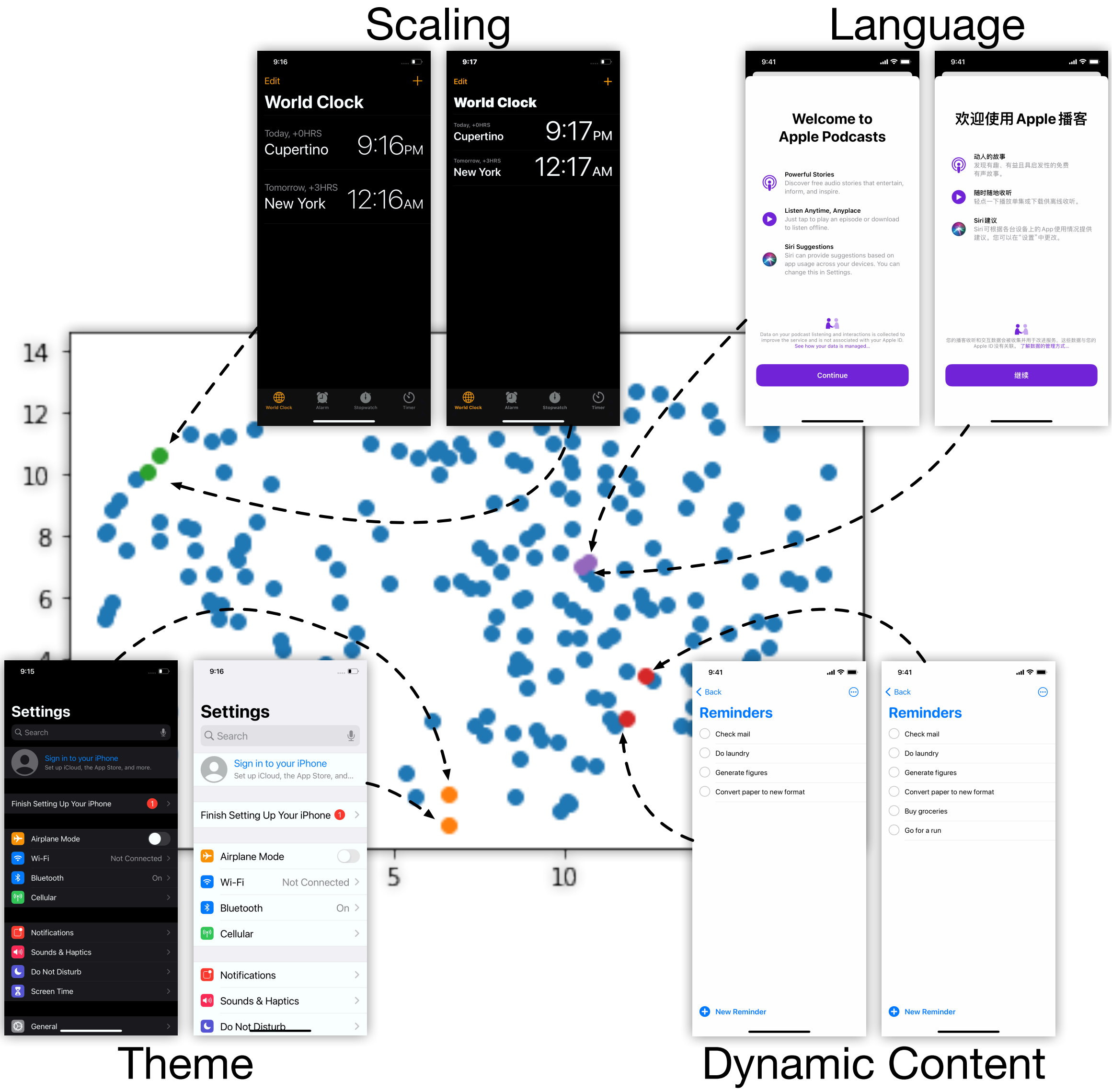

Object Detection Errors
Get Started
Collaboration
Edit with others at the same time on your Mac, iPad, iPhone, or iCloud.com.
Apple-Designed Templates
Quickly create beautiful reports, resumes and more.
Easy Formatting
Make text and images look great with preset styles.
Powerful Reviewing Tools
Use change tracking, highlights, and comments.
Documents
Create New
Incorrect Grouping
9:18
Music
Try it now and get 3 months free.
THURSDAY, JANUARY 23
For You
Let us find your next favorite.
Discover handpicked playlists & albums based on the music you love. Download and play millions of songs, even offline.
Try It Now
Not Playing
Library
For You
Browse
Radio
Search
Missing Grouping
10:46
Browse
Cloud Drive
Cancel
Search
Sorted by Name
Downloads
2 items
GarageBand for iOS
1 item
2 items, 5 GB available on iCloud
Recents
Browse

Figure 5.6: Examples of some errors by our screen parsing model. We identified three types of errors that can occur: *(i)* object detection errors, *(ii)* incorrect groupings, and *(iii)* missing groupings. Object detection errors can lead to missing elements or misaligned bounding boxes, which our model relies on to infer grouping. Incorrect groupings can assign irrelevant text labels to icons. Missing groupings can result in errors in downstream applications, such as a non-optimal navigation order for screen readers.



Welcome to Apple Podcasts

Powerful Stories
Discover free audio stories that entertain, inform, and inspire.

Listen Anytime, Anyplace
Just tap to play an episode or download to listen offline.

Siri Suggestions
Siri can provide suggestions based on app usage across your devices. You can change this in Settings.

Data on your podcast listening and interactions is collected to improve the service and is not associated with your Apple ID. See how your data is managed...

欢迎使用Apple播客

动人的故事
发现有趣、有益且具启发性的免费有声故事。

随时随地收听
轻点一下播放单集或下载供离线收听。

Siri建议
Siri可根据各台设备上的App使用情况提供建议。您可以在“设置”中更改。

您的播客收听和交互数据会被收集并用于改进服务，这些数据与您的Apple ID没有关联。了解数据的管理方式...

Figure 5.7: The intermediate representation of our parsing model can be used to produce a screen embedding, which describes hierarchical structure of an app. We embedded a set of app screens using our model and visualize them in a 2-D projection. We show that display settings such as *(i)* scaling, *(ii)* language, *(iii)* theme, and *(iv)* small dynamic changes result in minimal variation, which may be useful for some downstream tasks that rely on characterizing screens by semantic structure rather than aesthetic appearance.

5.7.2 Accessibility Enhancement

Screen readers help blind and visually impaired users access applications by reading out content highlighted by a cursor. Knowledge of UI element location (*i.e.,* spatial information) and hierarchy is important for screen readers to compute the correct order to move the cursor through the screen (*e.g.,* elements in the same group should be ordered consecutively), and for accessible apps, this information is found in an app's accessibility metadata. Recent work [347] has successfully generated missing metadata for inaccessible apps by running an object detection model on the UI screenshot. Their approach to generating hierarchical data relies on manually defined heuristics that detect and group localized patterns between elements (*e.g.,* a container with a text element inside it might be grouped as a button). However, these approaches may sometimes fail because they do not have access to global information that is necessary for resolving ambiguities.

In contrast, our implementation generates a UI hierarchy with a global view of the input, so it can overcome some of the limitations of heuristic-based approaches. We used the predicted UI hierarchy to group together the children of intermediate nodes of height 1 that contained at most one text label and used the X-Y cut algorithm [226] to determine navigation order. Figure 5.8 shows an example where the grouping output from the screen parser model is more accurate than the one produced by Screen Recognition heuristics. Note that this is not always the case. More examples are available in the appendix.

5.7.3 Generating UI Code from a Screenshot

Producing code from screenshots or mock-ups can greatly accelerate application prototyping development. A simple approach for code generation is *(i)* to first extract the location and type of UI elements using an object-detection model, *(ii)* then generate code that places the appropriate UI controls at the detected locations. While this approach may result in interfaces that are visually similar to the input, it is undesirable for several reasons. Code generated using this approach often uses absolute positioning constraints to instantiate UI controls, which prevents it from adapting to new screen sizes and makes it less useful for developers to work off of.

Some systems [238] use heuristics to detect a limited subset of containers (*e.g.,* lists), while others [57] augment visually detected elements with hierarchical data extracted from the window manager. To generate high quality, responsive code, structural understanding of a UI is an important step.

We built an example application that uses our implementation to generate SwiftUI code from a app screenshot. We employed a technique used by compilers to generate code from abstract syntax trees (AST) known as the *visitor pattern*. First a screenshot is fed into our system, which produces a UI hierarchy. We performed a depth-first traversal of the UI hierarchy using a *visitor* function that generates code based on the current state (current node and stack). Specifically, the visitor function emits a SwiftUI control (*e.g.,* Text, Toggle, Button) at every leaf node and emits a SwiftUI container (*e.g.,* VStack, HStack) at every intermediate node [2]. We manually created a mapping between nodes types in the screen parser tree and SwiftUI views and automatically required parameters such as label text using OCR. Elements containing graphics, such as image

[2] Information about SwiftUI controls and containers are available in the SwiftUI documentation: `https://developer.apple.com/documentation/swiftui/views-and-controls`

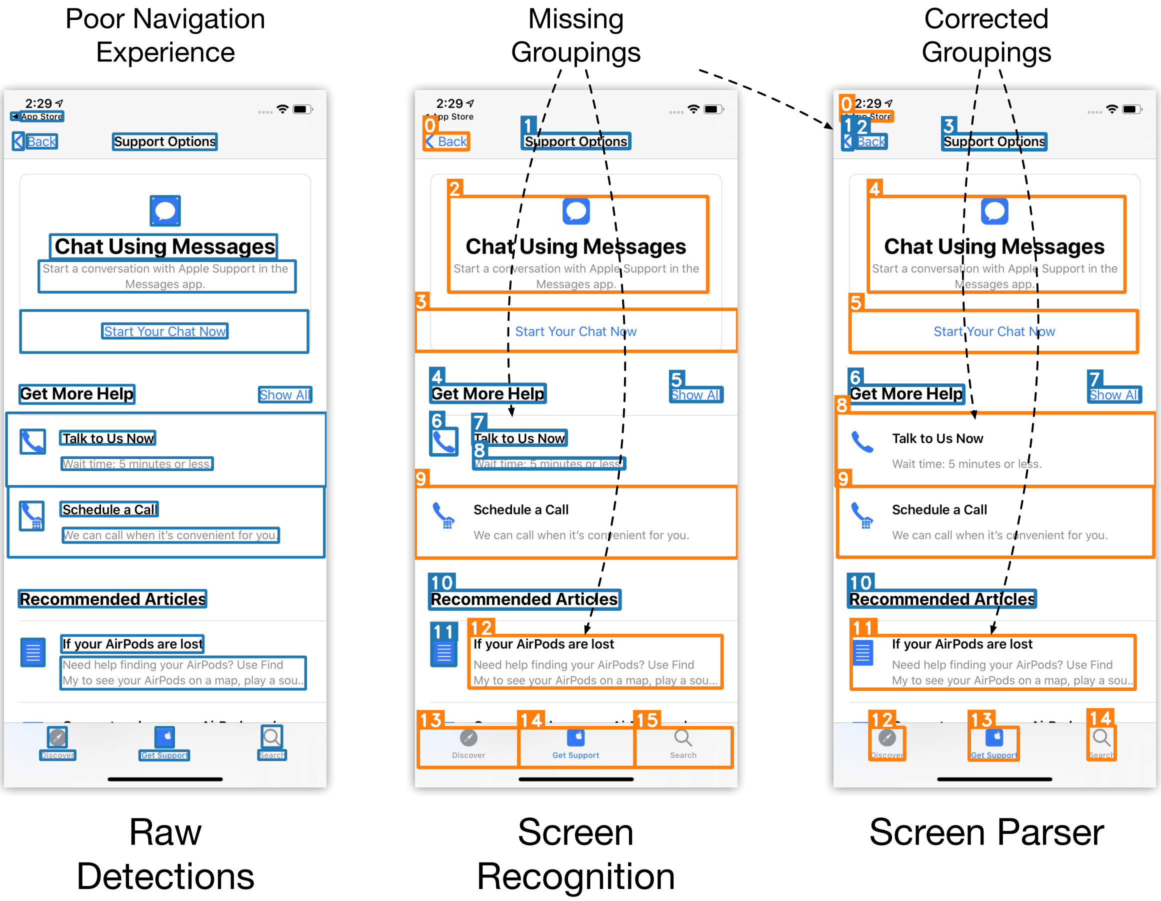


Figure 5.8: Recent approaches use object-detection approaches to generate accessibility metadata for inaccessible apps. Our model can be used to improve or augment the heuristic-based approach used by these systems to infer navigation order. Original detections from the object detector are shown in blue, and grouped elements are shown in orange. Element boxes are annotated using their navigation ordering [226], where the number represents how many swipes are needed to access the element when using a screen reader. While both results contain errors, in this case, Screen Parser correctly groups more elements, which decreases the number of swipes needed to access elements.

views or icons, are represented by an image patch cropped from the original screenshot, which are automatically included as assets. When generating code for small form-factors such as smartwatches, we replace horizontal containers with vertical ones due to limited space. Finally, our system uses a simple heuristic to determine whether the app uses a light or dark theme, and sets a preferred color scheme.

The resulting code describes the original UI using only relative constraints (even if the original UI was not), allowing it to act responsively to changes in screen size or device type (Figure 5.9). The generated code does not contain appearance and style information (*e.g.,* text size, element color), which is sometimes necessary to render a similar-looking screen. Nevertheless, prior work [61] has shown that such output can be a useful starting point for UI development, and we believe future work can improve upon our approach by detecting these properties.

5.8 Limitations and Future Work

In this paper, we presented the problem of *screen parsing* and implemented a baseline implementation that shows how structured information can be predicted from a UI's visual appearance. Specifically, our implementation predicts the *presentation model* from a UI's screenshot, for which we had a large dataset of examples (*i.e.,* view hierarchies) to facilitate machine learning. Some of our system's constraints (*e.g.,* can only produce directed trees) were purposefully introduced by us in service of our chosen target model.

We see multiple opportunities to improve our particular implementation. From our evaluation, we found that certain components, such as the UI Element Detector, can limit the performance of others that rely on it. The performance of our system can also be improved by modeling changes *e.g.,* incorporating visual information (*e.g.,* dominant color or visual embedding of an element) in our hierarchy prediction and improving our group labeling model. Some down-stream applications have different notions of performance. For example, when computing screen reader navigation, lower-level groupings (*i.e.,* close to leaf nodes) matters more. To more accurately assess our system's performance, we plan to evaluate it in the context of down-stream tasks. Our current model generates its output entirely from visual input (i.e., app screenshot), which minimizes its dependencies. Nevertheless, we also believe there is an opportunity to take advantage partial or incomplete view hierarchy, which our model can use in conjunction with visual information to improve performance [57].

More broadly, we hope to apply *screen parsing* to extract other types of structured semantics from UI screenshots (including those on other platforms such as web and desktop UIs), including those that describe data flow, interaction, and tasks. We expect that for some of these, we will be able to re-use much of our current architecture. Others might require adding or moving constraints (*e.g.,* predicting more general types of graphs that may include cycles). Furthermore, some types of models (*e.g.,* task models) might not be possible to infer from a single screenshot and would instead require a sequence of screens. Regardless, we expect the utility of automated UI systems to increase as they gain the ability to parse reason about structured semantics from UIs. We believe a promising application of screen parsing lies in tasks that require higher-level semantics such as task automation and programming-by-demonstration [191], which often require accessibility metadata to work.

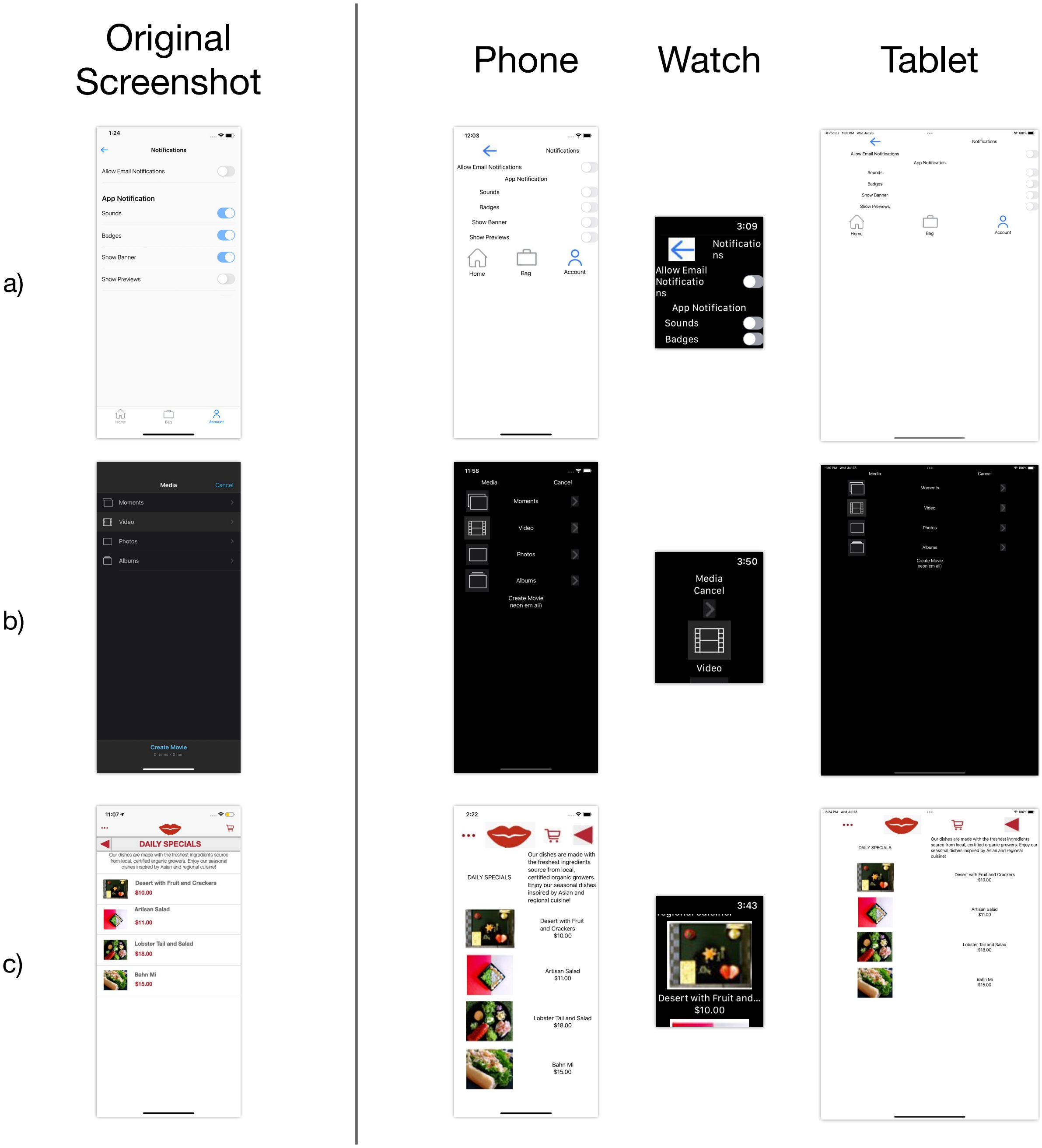


Figure 5.9: By mapping nodes in the UI hierarchy to declarative view-creation methods, we can generate code for a UI from its screenshot. Generating code from the hierarchy rather than the layout ensures that it is responsive across screen sizes, and we show the same output code rendered on different device form factors. Our example application may produce some errors due to missing style information (a, c) or inaccurate OCR (b).

5.9 Conclusion

In this paper, we introduced the problem of *screen parsing*, the prediction of structured UI models from visual information. In a comparison to three related problems, we show that our problem formulation and model is more suited to the unique properties of user interfaces. We described the architecture and training procedure for our reference implementation, which predicts an app's presentation model as a UI hierarchy with high accuracy, surpassing baseline algorithms and training procedures. In addition, we showed that the properties of our system allow it to simultaneously support a diverse array of down-stream applications: *(i)* UI similarity search, *(ii)* accessibility enhancement, and *(iii)* code generation from UI screenshots. More broadly, we believe our formulation of *screen parsing* will allow automated systems to better reason about the underlying structure and purpose of UIs, facilitating more advanced and complex interactions.

5.10 Model Hyperparameters

Model	Hyperparameter	Value
Faster-RCNN	optimizer	SGD
	lr (base)	0.01
	lr (max)	0.1
Screen Parser	optimizer	Adam
	lr	1e-4
	weight decay	1e-5
	dropout	0.25
	hidden size	256
	hidden layers	4
Group Labeler	optimizer	Adam
	lr	1e-4
	weight decay	1e-4
	hidden size	256
	hidden layers	1

All models were trained with early stopping that stopped training when validation loss did not improve for 10 epochs. We implemented our models using PyTorch [254] and PyTorch Lightning [97].

5.11 Oracle Pseudocode

```
for example in dataset:
    node = example.rootNode
    while not example.isTerminalNode(node):
        optimal = set()
        if len(node.children) == 0:
```

```
        # at a leaf node, go back up
        optimal.add(PopAction())
    else:
        for child in node.children:
            if child.isLeaf:
                optimal.add(ArcAction(child))
            else:
                optimal.add(EmitAction())
    if dynamicTraining:
        model.train(optimal)
        if model.highestScoring in optimal:
            action = model.prediction
        else:
            action = randomChoice(optimal)
    else: # static training
        action = getCanonicalAction(optimal)
        model.train(action)
    node = node.nodeAfterAction(action)
```

5.12 UI Retrieval Examples

Figure 5.10 shows examples of UIs retrieved using our UI Similarity Search example application. We embedded a set of query UIs and used them to retrieve similar UIs from a subset of our AMP dataset.

5.13 Accessibility Enhancement Examples

Figure 5.11 shows examples of accessibility metadata generated by our system and Screen Recognition, a baseline that we compared with. Both systems occasionally produce minor errors (*e.g.,* grouping elements that do not belong together) but significantly improve the navigation experience.

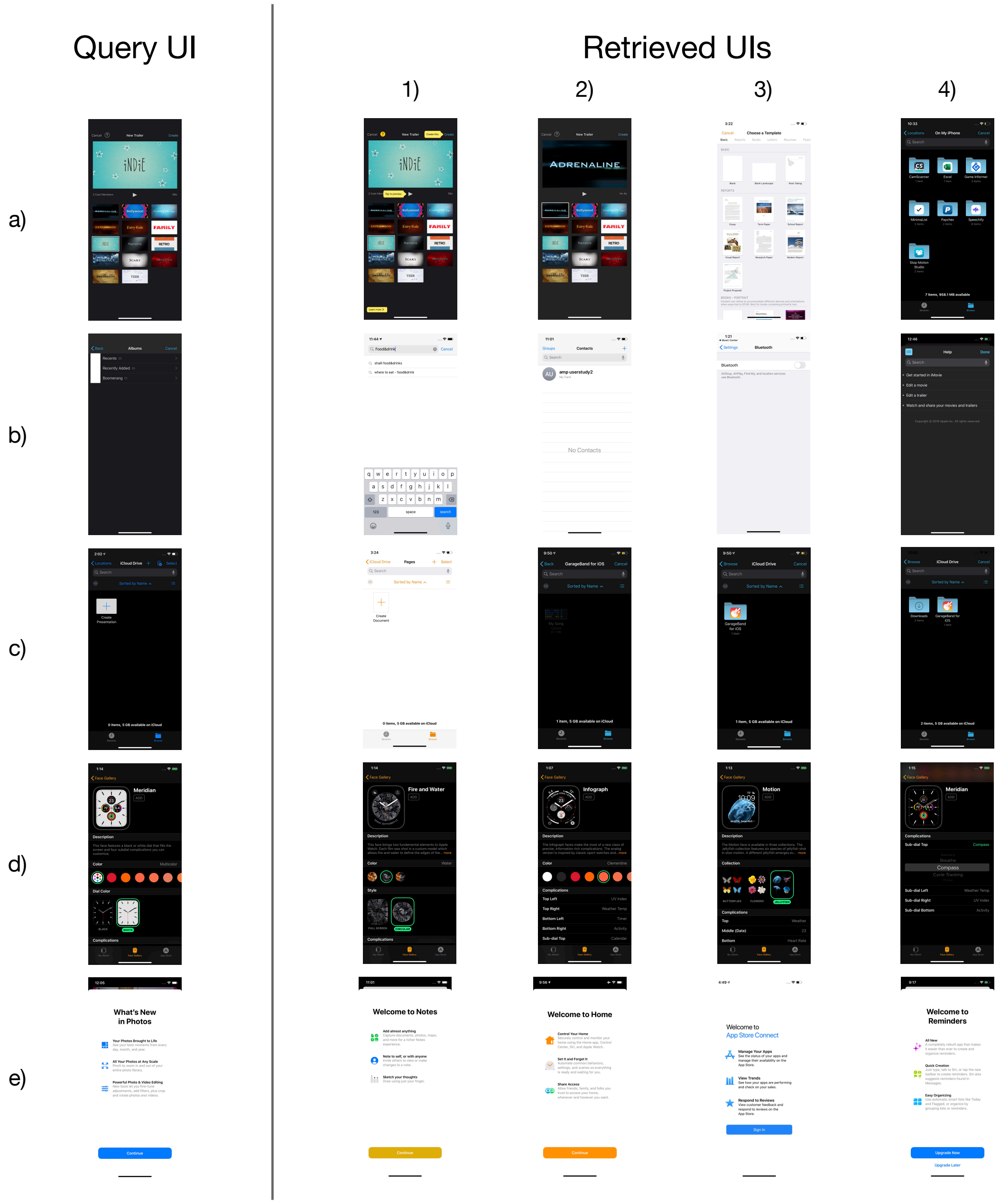


Figure 5.10: Example output of our UI Similarity Search example applications. We use several query UIs to find similar UIs in a subset of the AMP dataset. Retrieved UIs are ordered by their similarity to the query UI in embedding space. Many of the retrieved screens are from other apps with similar structural layout.

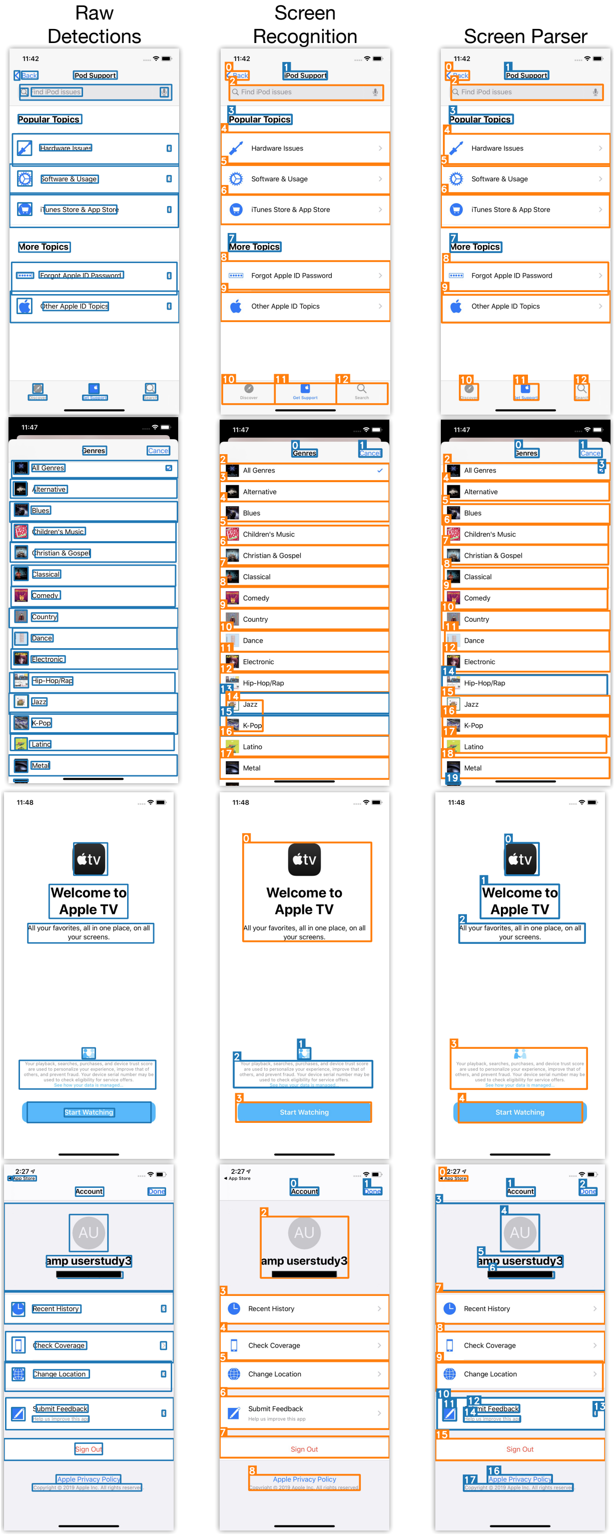


Figure 5.11: Examples of accessibility metadata generated for raw detections by Screen Recognition heuristics and our screen parser model. Each element is annotated with the number of swipes needed to reach it using a screen reader. Elements groups are shown in orange. The last row of screenshots contain an email address, which is redacted.

6
Never-ending Learning of User Interfaces

6.1 Preamble

Content in this chapter is based on a conference paper that was published at UIST 2023 [328].

This chapter presents a system with two goals. First, similar to the preceding two chapters, we developed a model to predict a new type of semantic about UIs. We focus on affordance prediction *i.e.,* understanding which UI elements can be interacted with using taps or other gestures, such as dragging. This is essential for machine-based understanding of UI functionality, which is ultimately necessary for context-dependent transformations. Second, we present a novel approach for training ML models to learn this information. Currently, most models rely on datasets of static screenshots that are labeled by human annotators, a process that is costly and surprisingly error-prone for certain tasks. For example, workers labeling whether a UI element is "tappable" from a screenshot must guess using visual signifiers, and do not have the benefit of tapping on the UI element in the running app and observing the effects. In this paper, we present the Never-ending UI Learner, an app crawler that automatically installs real apps from a mobile app store and crawls them to infer semantic properties of UIs by interacting with UI elements, discovering new and challenging training examples to learn from, and continually updating machine learning models designed to predict these semantics. The Never-ending UI Learner so far has crawled for more than 5,000 device-hours, performing over half a million actions on 6,000 apps to train three computer vision models for i) tappability prediction, ii) draggability prediction, and iii) screen similarity.

6.2 Introduction

Machine Learning (ML) has played an increasingly important role in the domain of mobile User Interfaces (UIs). Recent techniques have used Deep Neural Networks (DNNs) to bridge critical usability gaps and enable new types of evaluations, such as providing missing accessibility metadata to UIs [325], giving designers feedback to make UI features more discoverable [278, 295], and predicting user engagement with animations [333]. The enabling research artifacts behind these interactions are large datasets of mobile UI screenshots annotated by human annotators [81, 171]. These datasets provide an invaluable volume of data for training DNNs, but they only capture a fixed snapshot of the views of mobile applications and are extremely costly to collect and update. In addition, relying on annotators to estimate certain properties of UI elements from static visual signifiers is known to be error-prone [278]. Inspired by the Never Ending Learning paradigm [228], we propose an automated method for collecting UI element annotations by *interacting with applications directly* with an automated crawler that continuously improves its own performance and can refresh ML models for other downstream tasks over time.

We built the *Never-ending UI Learner*, an app crawler that formulates UI semantic learning as an active process that uses real interactions on real devices to explore UIs and discover properties which are used to continually train machine learning models. More specifically, our crawler automatically installs real apps from mobile app stores and crawls them to discover new, challenging training examples to learn from (e.g., those that result in low model confidence). During crawling, the Never-ending UI Learner records temporal context (i.e., taking screenshots before, during, and after interactions) that is used by heuristic functions to generate more accurate labels than are possible from human-annotated single screenshots. The resulting data is used to train models that predict the tappability and draggability of UI elements and determine the similarity of encountered screens. Although the process can start with a model trained from human-labeled data, the end-to-end process does not require any additional human-labeled examples.

In contrast to existing data pipelines for data-driven UI modeling [81, 171, 346], our never-ending UI learning paradigm allows data collection, annotation, and model training to be performed without any human supervision and can be run indefinitely. Of course, in this paper the learning is not truly never-ending. Here we present experiments that analyze the performance characteristics of our learner over 5,000 device-hours, in which it performed more than half a million actions on 6,000 apps. The resulting dataset is an order of magnitude larger than existing human-annotated UI datasets [81, 346] and allowed us to analyze the performance of UI semantic models when trained with increasing amounts of recently collected examples. Ultimately, we believe this model can be used in a true “never-ending” style, continually crawling the app ecosystem, collecting data from literally all available apps, and experiencing new UI styles and trends as new or updated apps are released.

The specific contributions of our paper are as follows:

1. **The Never-ending UI Learner**, is a system that operationalizes our approach for automatically learning from UIs through never-ending interaction.
2. **Three applications which demonstrate use of the Never-ending UI Learner**. We use our crawler to train three types of models of UI semantics that are difficult to learn through existing methods: i) tappability, ii) draggability, and iii) screen similarity.

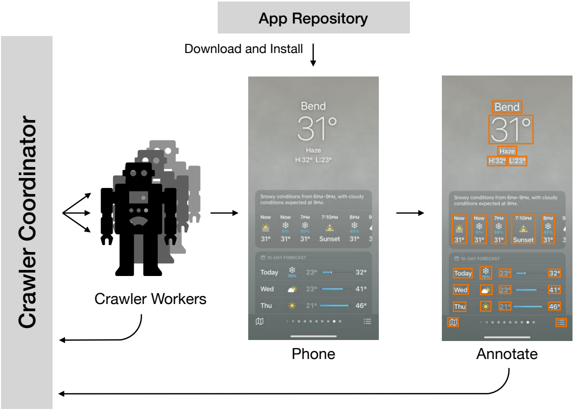


Figure 6.1: Architecture of our Never-ending UI Learner. The Never-ending UI Learner is a parallelizable mobile app crawler which consists of a coordinator-worker architecture. The crawler coordinator distributes crawls to workers and maintains the dataset. Each crawler worker is connected to a programatically controlled mobile device which collects data and runs data post-processing.

6.3 Never-ending UI Learner

To operationalize our approach, we built the Never-ending UI Learner, a system that automatically downloads and crawls publicly available apps using remotely operated devices. Our current implementation and infrastructure is based on iOS. We use stock factory reset devices that are logged in to testing accounts that are not associated with any real user data to avoid privacy concerns.

Note that unlike some crawlers that interact with apps using an OS-provided programmatic interface such as the accessibility API, our crawler interacts with the device through the VNC remote desktop protocol, from which it receives regular updates to the screen and processes them visually and can send raw input events to the device to create tap, swipe and keyboard actions. Using VNC, the Never-ending UI Learner is able to reliably interact with more apps, learn based on the same facilities that a human would and generalize to other platforms. In this section, we describe the crawler's architecture and behavior that enable it to perform never-ending learning.

6.3.1 Architecture Overview

Our crawling architecture is shown in Figure 1. We implemented a distributed crawling architecture which consists of i) a central coordination server and ii) a large pool of workers to parallelize the crawling process.

Coordinator Server

The crawler coordination server maintains a list of app IDs to crawl which are sent to workers. The central server keeps track of successful and unsuccessful crawls, and it automatically retries failed app crawls. App crawlers differ from web crawlers in that they focus only on the app they are asked to crawl, although limited cross-app interaction sometimes does occur (e.g., clicking on a link or permission request dialog). When all app IDs are exhausted, our crawler can schedule itself to be run again after a fixed time period (e.g., weekly). The list of app IDs can be modified between crawls to add new apps or reflect changes in app availability. While the majority of the app IDs remain the same, the apps may change their appearance and behavior due to dynamically updated content and new versions of the software. Re-crawling the same apps regularly can enable our model to adapt to design changes over time.

Crawler Worker

Crawler workers are processes that interface with remotely controlled mobile phones and process the collected data. Each crawler worker downloads and installs a target app whose ID is provided by the central server to the mobile phone and then runs a program that crawls the app. Screenshots are collected during interactions and when the crawler believes it has arrived at a new screen. The program can use three methods to explore the app (random selection or based on model confidence), and as a part of this paper, we run experiments to determine the best crawling strategy for each of our never-ending learning use-cases. We set a time-limit (5 minutes) on the maximum duration of a crawl for a single app. Afterwards, the worker processes the collected data (e.g., screenshots and interactions) with models and heuristics to generate labels from the observations. Both raw data and processed output is uploaded to a coordinator server. In our experiments, the number of crawler workers varied to due to availability from the device pool which we used, which was shared with other users. Generally, there were around 40-100 crawler workers.

6.3.2 Machine Learning Components

Our crawler contains a screen-level and element-level model that allow it to understand the content on UIs it encounters. We run these models every time a screenshot is captured to augment it with useful semantics. Furthermore, the three UI semantic models that we trained in using the crawler, are designed as extensions of these base models, improving overall efficiency.

Screen Understanding

To keep track of its crawling progress in the app, our crawler uses a model to generate semantic representations of screens. We used a model introduced by previous work [101] that predicts whether two screenshots belong to the same UI by encoding each as an embedding vector, which the authors shared with us. Because significant variation can be introduced by changes in state, such as a news app that displays new content periodically, the model is designed to learn the underlying structure of UIs. We made minor modifications to the previous work in order to develop a model that could run under our hardware constraints. Instead of their recommended *screen transformer* model architecture, we use their CNN-based model architecture, which is more efficient to run despite somewhat lower performance [101]. For further optimization, we use an EfficientNet-B0 [299] model architecture as the backbone instead of the original ResNet-18 [133], which has more parameters. As in the original paper, the output of the last layer of the

CNN network is used as a screen embedding. During training, we applied a data augmentation approach [303] to increase performance. We followed all other aspects of the original model training and our final CNN-based model achieves a F1-score of 0.636.

Element Understanding

To generate element semantics, we used an object detection model architecture that is similar to CenterNet [352]. At a high level, the detection model slides a window (via convolutions) over the image and featurizes image sub-regions using a backbone network (MobileNet-v1 [142]), resulting in embeddings for each region. These embeddings are fed into a classification head which produces per-class confidences, and regions with high confidences are returned as detections. The model was trained on the AMP dataset [346], which consists of 77,000 app screens collected and annotated by annotators from 4,000 iPhone apps. In addition to the standard element type classification head, which was trained with the rest of the object detection model, we added heads for tappability prediction and draggability prediction. The additional heads are trained independently from the rest of the model by first freezing the backbone and training the heads on embeddings corresponding to detected elements.

6.4 Applying Never-ending Learning

In this section, we describe the application of our never-ending learning framework to three UI semantic models: i) tappability prediction (element semantic), ii) draggability prediction (container semantic), and iii) screen similarity (screen semantic). The tappability and draggability models were trained completely from crawler-generated data, while the crawler fine-tuned its existing screen similarity model that was originally trained using human-annotated data. For each UI semantic, we developed an interaction-based heuristic used by our crawler to automatically generate new training examples for our models. Next, we designed and trained models to predict each of these semantics from a screenshot. Finally, to contextualize these models in the context of never-ending learning, we analyzed their performance over time.

Experimental Setup. We conducted experiments on a list of 6,461 free iOS apps. For the purposes of evaluation, all model training and experiments were performed with randomized training (80%), validation (10%) , and testing (10%) splits. We randomly partitioned our list of app IDs, which ensured that all UI screens from an app were contained in the same split. We use the term *crawl epoch* to refer to one complete pass through the list of apps. Note that unlike an *epoch* through a training dataset, the actual contents of a *crawl epoch* might change from time to time, due to the dynamic nature of apps.

Our experiments analyzed two aspects of the crawler's performance: i) crawling strategy and ii) performance over time. We ran three variations of the crawler, which had different crawling strategies: i) randomly selecting elements on each screen (Random), ii) selecting elements that result in low prediction confidence from the current models (Uncertainty Sampled), and iii) a hybrid that for each crawl epoch alternates between Random and Uncertainty Sampled strategies, inspired by similar approaches in optimization [288]. To evaluate the performance over time, we ran each crawling strategy for five crawl epochs. Note that the first crawl epoch for all strategies uses Random to train an initial confidence-prediction model. In the Hybrid strategy, because alternation happens at the epoch level, the second epoch is crawled using the Uncertainty Sampled strategy and thus through two epochs the inputs and results are identical for both the Uncertainty Sampled and Hybrid strategies. The three strategies fully diverge starting from the third epoch. Across all experiments, we collected over half a million screenshots, although the same UI screen may have been visited multiple times. The number of screenshots in our dataset is considerably larger than previous work [53, 81, 171, 329].

The crawler's models were trained and evaluated after each crawl epoch. After each crawl epoch, model training is resumed with the updated data from the latest crawl and the model weights are optimized for 100,000 optimization steps (with early stopping). In order to maintain a constant validation set across a varying number of epochs, we only use the evaluation data split from the first epoch for calculating performance metrics. Finally, for models that were trained completely on crawler data (tappability and draggability), we performed additional sub-epoch evaluations during the first crawl to analyze learning speed.

While the dataset is not released at the time of publication due to internal regulations, we are investigating processes to make it available to the broader community. To replicate our work, it is possible to use tools and models built for comparable platforms (e.g., Android). Open-source crawlers [199, 309] can be integrated with available screen similarity [329] and element detection models [53, 329, 334].

6.4.1 Tappability

Tapping is the most common interaction on mobile devices, yet it is often difficult to automatically determine if an element is tappable or not due to missing metadata and ambiguous visual cues. For example, a text button that doesn't have sufficient contrast or missing borders would likely appear untappable to users, and many games are missing accessibility traits that prevent screen reader users from using them. Accurate inference of tappability could aid designers in finding ambiguous visual elements and be useful for generating metadata for repairing inaccessible apps. Previous work has used human-annotated UI screenshots to train machine learning models of tappability. However, this process is surprisingly error-prone [182, 186, 278, 295] due to ambiguous visual cues, which suggests that human-annotated screenshots are an unreliable source of ground-truth for training tappability models. In contrast, our crawler can use additional context from the entire interaction, such as before and after screenshots instead of a single before screenshot, to determine if tapping resulted in an effect. Effects could either be state changes, like flipping a toggle, or a transition to a new screen. We developed a heuristic for inferring tappability from our crawler's recorded interactions and found that it had high agreement with human-annotated videos. We used heuristic-labeled data to train an efficient tappability "head" model purely from crawler-annotated data. After five crawl epochs, the best-performing tappability model reached an F1 score of 0.860.

Tappability Heuristic

We developed a heuristic to infer the tappability of an element based screenshots of the UI taken before, during, and after a tap interaction. A tap may result in several different scenarios, which

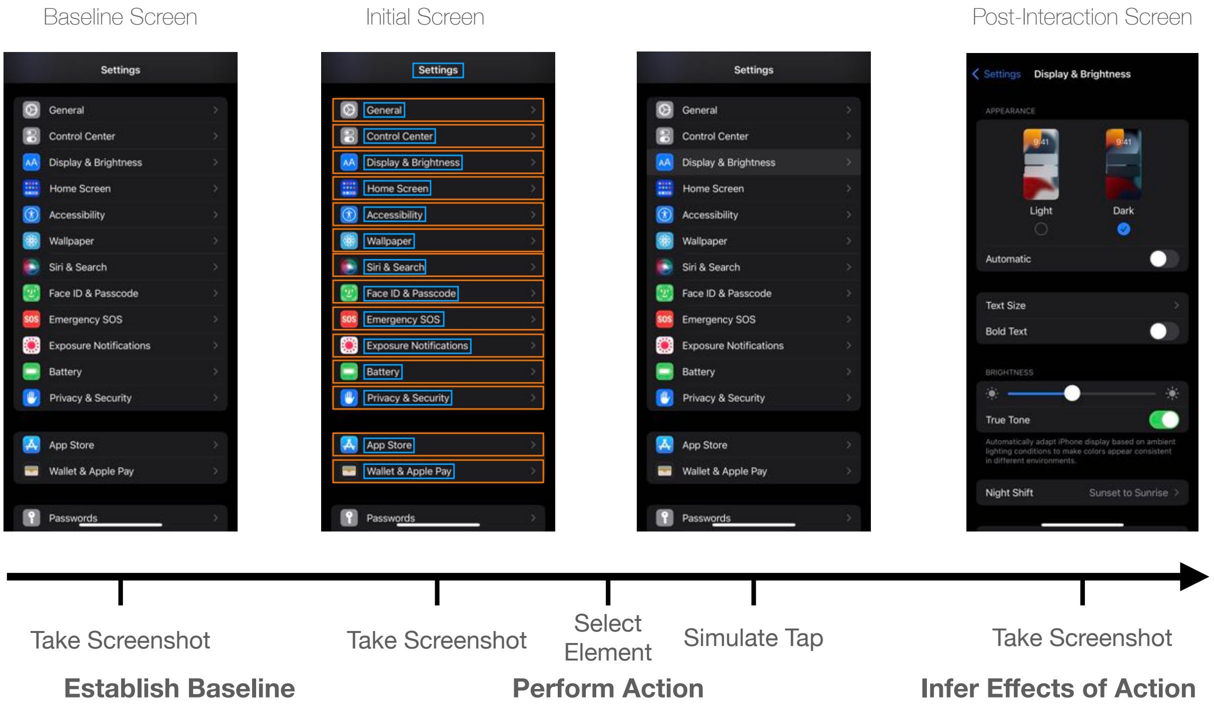


Figure 6.2: This figure visualizes the steps to our tappability heuristic. When the crawler arrives at a new screen, it takes two screenshots separated by 5 seconds as a baseline of visual change. Then, a detected UI element is chosen and sent a tap. After waiting for the screen to settle, a post-interaction screenshot is used to infer the effects of the action.

are captured by our heuristic. First, we use a screen similarity model to compare screenshots taken before and after the tap to determine if the tap led the crawler to a new screen. If a screen change was not detected, the tap could have also changed the screen state. We compute a pixel-based difference of the "before" and "after" screenshots to identify possible visual indications of local or global changes, such as tapping a checkbox or refreshing screen content respectively. Finally, to reduce false positives, the heuristic also uses multiple screenshots captured before the tap to identify dynamic areas of the screen (e.g., videos) whose visual changes are not related to the tap.

To validate the accuracy of our heuristics, we compared its results against human-labeled interaction videos. We used our crawler to save short screen recordings of tap interactions that were collected during crawls. Each example video was approximately 10 seconds long and included the tap location overlaid on the video and temporal context before and after the tap interaction, such as including transition and loading animations.

We randomly sampled a balanced subset of 1000 video clips from our crawls and asked human annotators if each video clip contained a tap interaction. Annotators were recruited, trained, and paid by a separate team at our organization (all with appropriate legal/ethical approval). Annotators were employees paid who are paid a competitive hourly salary for their location. We used standard classification metrics to evaluate the accuracy of our heuristics, using the human-annotated labels as ground truth. The tappability heuristic had an overall accuracy of 0.934, and had a similar number of false positives (38 instances) and false negatives (28 instances).

Model Implementation

To predict tappability, we designed a model architecture that operates as a "head" of our existing element detection model (Figure 6.3). Heads are small sub-networks or set of layers usually located close to the output layer of neural network architectures and generate predictions from featurized representations of the main input produced by a "backbone" network. Since element detection is closely related to tappability, we hypothesized that the previously learned representations are likely to contain relevant information and greatly accelerate tappability learning. Our head model is a simple three-layer feed-forward model with an input size of 128, a hidden size of 64 that we chose through manually tuning, and an output size of 1 that gives tappability confidence. To train it, we first froze the weights of the element detector's backbone network and randomly initialized the parameters of our feed-forward network. While freezing most of the model reduces its capacity, it also results in a significant reduction in training time, since there are fewer parameters to optimize. Then we trained the model to predict the tappability of an element from a screenshot of the UI before the tap, and we used the labels generated by our tappability heuristic as ground truth.

Performance Evaluation

The results of our experiments are shown in Figure 6.4. While all crawling strategies are successful in improving on the initial model from the first epoch, the Random crawler has the best final performance. In our experiments, the Random crawler reaches the best final F1 score of 0.860 while the Uncertainty Sampled crawler reaches the lowest final F1 score of 0.853. While it is not

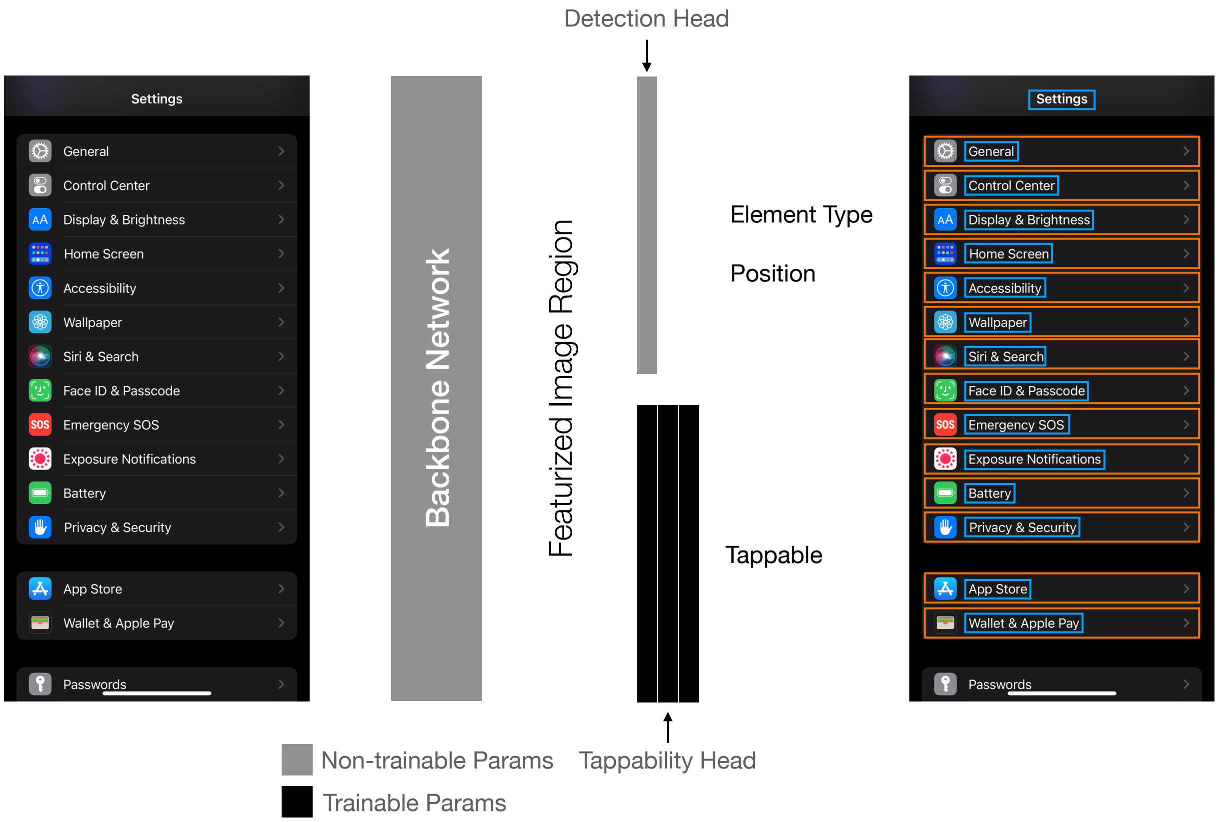


Figure 6.3: Architecture of our tappability model. The tappability model is designed as a "head", which is a sub-network of the UI element detection model. The element detector featurizes image regions in an input screenshot using a sliding window, which results in a featurized image embedding for each detected object. The main branch of the network (top) feeds in the embedding to determine the region's element type and position. We feed in the same element embedding into a separate feedforward network (bottom) to predict the probability that it is tappable.

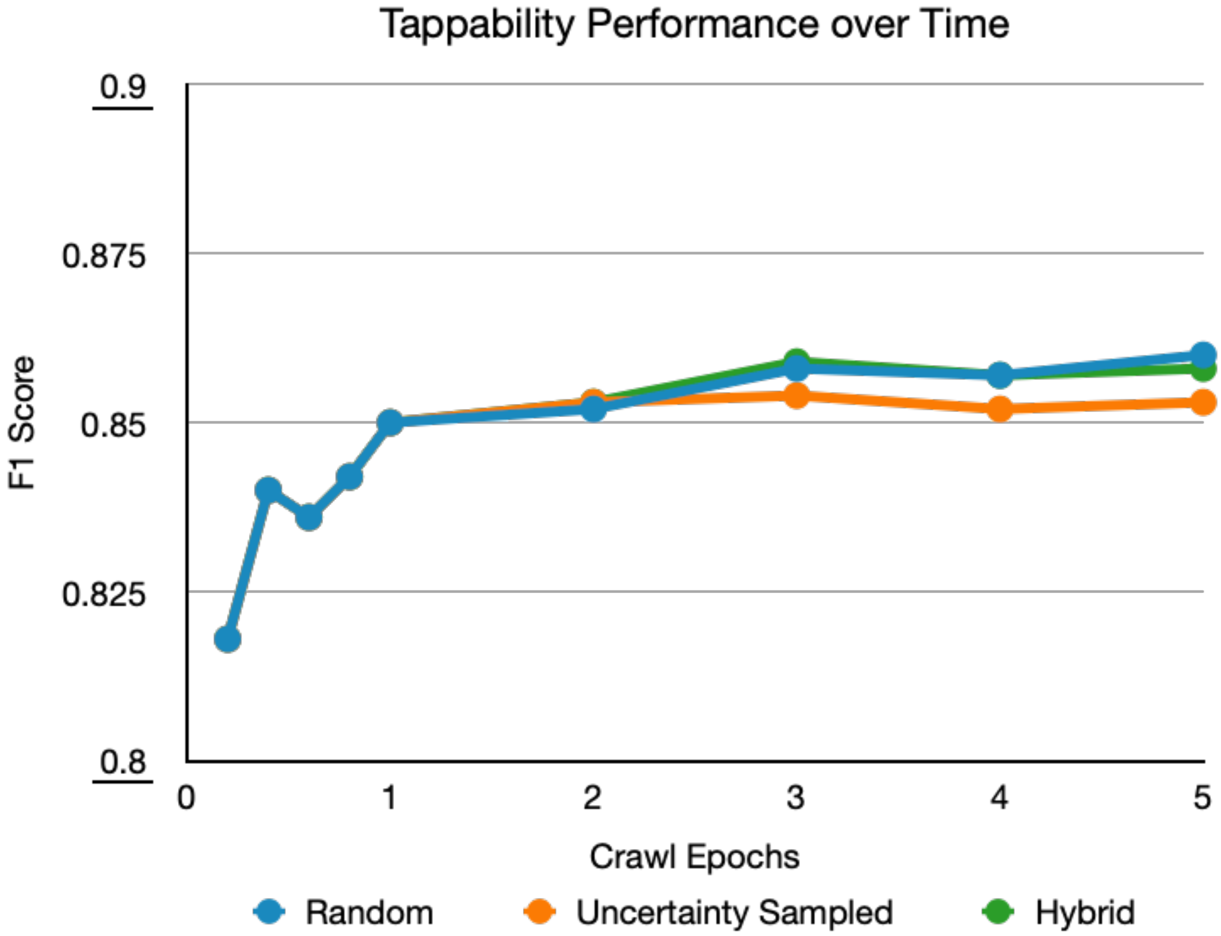


Figure 6.4: Performance of tappability over time. The model performance increases most rapidly during the first crawl epoch and the rate of improvement plateaus afterward. After the final epoch, the random crawler achieves the highest F1 score of 0.860, and the uncertainty sampled crawler has the lowest F1 score of 0.853.

possible to make a direct comparison with previous work [278, 295] because their experiments were run on different datasets, it seems that our tappability model is able to reach similar levels of performance in terms of F1 score after its first epoch.

We also conducted a comparison between the quality of our automatically collected tappability dataset and human-annotated ones, we used the labels provided by the AMP dataset [346]. First, we trained our classification head model architecture on AMP, which led to similar performance (F1=0.81) to the originally reported numbers (also F1=0.81), which used a tree-based model architecture. However, when we used the model trained on human-annotated data to predict the tappability of elements in our crawled dataset, we observed significantly degraded performance (F1=0.60), suggesting that the human-annotated and crawler-generated labels disagree with each other. We consider the heuristic-annotated data to be higher quality since its performance was validated by annotators with access to a video clip of the entire tapping interaction, and previous work [278] has shown predicting element tappability from a single screenshot leads to high variance among raters.

6.4.2 Draggability

Dragging is a common interaction in mobile apps that involves touching an element on the screen with one's finger and moving the finger along the screen's surface before finally releasing it. This interaction is used to manipulate controls, such as sliders and page controls, and is necessary for accessing off-screen content via scrolling. While these examples reflect different types of input, we collectively refer to all these actions as "draggability," since they involve similar physical movement. Unlike tappable elements, draggable elements often have fewer visual signifiers and are more difficult to automatically detect. To the best of our knowledge, there aren't any datasets available with draggability labels, and we believe that, similar to tappability, it would be difficult for human labelers to reliably identify draggable elements from screenshots. To improve screen reader support for inaccessible apps with these affordances, we used our crawler to automatically find and label examples through automated interaction. We developed a heuristic to infer draggability from screenshots of drag interactions. Using data labeled by this approach, we trained a draggability model that reached an F1 score of 0.794 after five crawl epochs.

Draggability Heuristic

To detect if a UI element is draggable, our crawler captures screenshots while attempting to hold and drag elements. Our crawler first identifies likely candidates, then emulates drag actions to the left (e.g., finger goes to the left) and upward directions to detect horizontal and vertical dragging, respectively. These directions were chosen because they correspond to the initial position of lists in left-to-right reading directions, and we execute drag actions from the center of the element to either its left or top boundary. The crawler captures one screenshot before the drag begins and one screenshot at the end of the drag but before its "finger" leaves the screen.

The high-level idea of the heuristic is to detect which UI elements, if any, "follow the finger" in the direction of the drag. We first use the smallest UI element containing the dragged pixel on the pre-drag image to create an image patch. This image patch is template-matched with

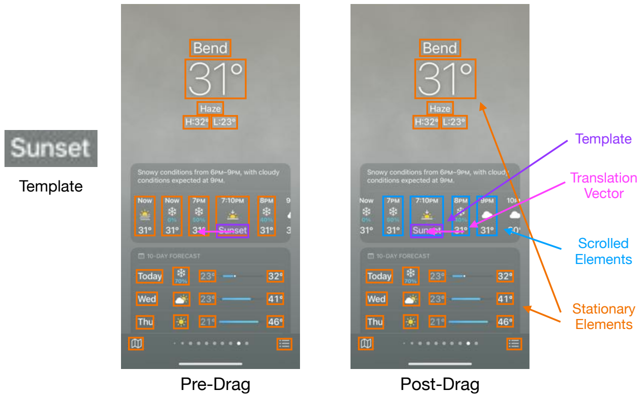


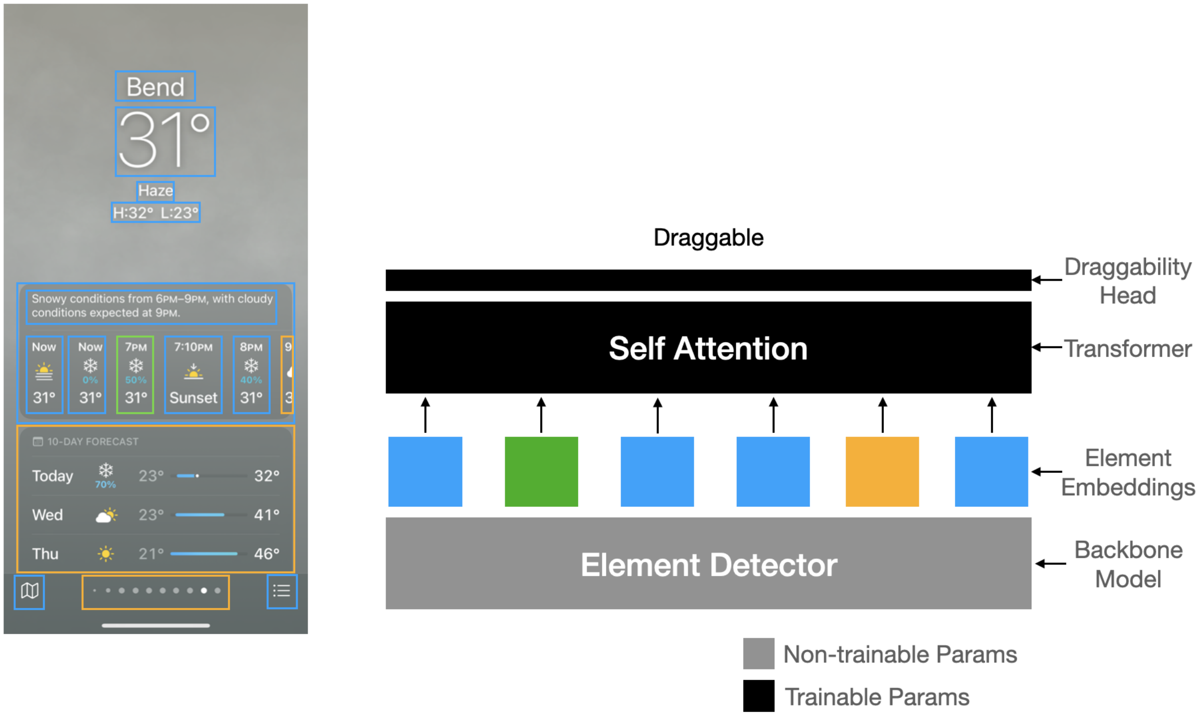
Figure 6.5: This figure illustrates the draggability heuristic. The heuristic uses a pre-drag image (center) which was taken before the interaction, and a post-drag image (right) which is taken near the end of the drag interaction, before the "finger" leaves the screen. A template image is created from the dragged element (left). The heuristic finds the location of the template in the post-drag image to infer draggability.

the post-drag image using the normed correlation coefficient method on grayscaled and edge-detected images. The vector corresponding to the template displacement is filtered by cosine angle and magnitude. Next, the patches inside bounding boxes between the pre-drag and post-drag screens are compared to identify whether other elements which scrolled during the drag action. If the contents of a bounding box in the pre-drag image match the contents of a bounding box *translated by the template translation vector* in the post-drag image, then it is likely a UI element which has been scrolled together with the original UI element. We use the normed correlation coefficient method to measure similarity between these image patches, on grayscaled and edge-detected images. If no scrolled elements are identified, the original UI element is also marked as not draggable to filter out false positives.

We conducted an evaluation of our heuristic on 1000 samples, which were generated by running the heuristic on screens collected from a randomized crawl, then selecting 500 screens where the heuristic was triggered and 500 where it was not. Due to a glitch, our crawler did not record the interaction videos of the draggability interaction, however we found that it was straightforward to infer draggability from the captured before/after screenshots. For each interaction step, we presented the annotator with three images, the pre-drag image, post-drag image, and a combined image with the both pre- and post- images super-imposed, which allowed more easy visualization of movement. The images were annotated with an arrow that indicated where the drag occurred. Again, we used the human-provided labels as ground-truth to evaluate our heuristic's predictions. The draggability heuristic had an overall accuracy of 0.92, and a similar number of false positives (38 instances) and false negatives (48 instances).

Model Implementation

Unlike tappability, which is an element semantic, draggability is often associated with containers that contain multiple elements. We initially tried to use the same "head" model architecture as our



Snowy conditions from 6PM–9PM, with cloudy conditions expected at 9PM.

▨ Non-trainable Params

■ Trainable Params

Figure 6.6: Architecture of our draggability model. Similar to the tappability model, the draggability model uses embeddings from the element detector. To give the draggability model additional context (e.g., presence of partially occluded elements), all elements on the screen are simultaneously fed into a single-layer transformer. The resulting contextual embeddings are used to predict draggability probability.

tappability model, which re-uses the element features generated by our detection model, however we found that this model achieved low performance (F1 score=0.2). Upon closer inspection of misclassified examples, we noticed that visual signifiers for an element's draggability are often non-local (i.e., occur elsewhere on the screen). For example, a picture is more likely to support swiping if page control indicator is located beneath it, and scrollability in mobile apps is often best inferred by searching for partially occluded elements at the end of the list or near the edges of the screen. Because the element detector featurizes image regions by pooling together nearby visual information, it omits many relevant cues for this task.

Based on these observations, we designed a model based on the transformer architecture, which allows it to incorporate information from the entire screen into its prediction. We first used our element detector to featurize all detected elements on the screen. The element embeddings are then fed into self-attention layers to generate a contextualized embedding. Finally, the elements' contextualized embeddings are fed into a linear classifier with a single output node to classify draggability. While training the draggability model, the element detector's weights are also frozen to improve training efficiency. For screens where the draggability heuristic wasn't triggered, loss is only computed for the directly interacted element. For screens where the draggability heuristic was triggered, loss is computed on all elements that were affected by the drag. In both cases, elements that did not move along with the finger are ignored in the loss calculation, as it isn't possible to know for certain if they are not draggable without interacting with them.

Performance Evaluation

Our evaluation of the draggability model focused on performance over time (See Figure 6.7).

The results of our experiments are shown in Figure 6.7. The Hybrid crawler had the highest final performance (F1=0.794), while the Uncertainty Sampled crawler was lowest (F1=0.770). Interestingly, the Uncertainty Sampled and Hybrid crawls both experienced a decrease in performance during the second crawl epoch. While the Uncertainty Sampled crawler continued to decline, the Hybrid crawler alternated to its randomized crawl strategy and began to rapidly improve. We hypothesize that the uncertainty sampling during the second epoch may have imbalanced the dataset by collecting many examples of similar elements while ignoring others, and thus negatively impacted the subsequent model.

From our experimental results and anecdotal observations, we hypothesize that draggability is harder to infer from static visual information alone due to the lack of local cues, and the best way to discover functionality that involves dragging may be learning from extended usage. In some cases, it may be appropriate to directly apply the draggability heuristic at run-time. In contrast to tapping, which is likely to alter the state of the UI or bring the user to a new page, we hypothesize that many dragging interactions are less likely to lead to side-effects. Our model could be used to first identify likely candidates for interaction-based verification.

Similar to the tappability model, we also observed small gains in performance over time; however, there was less overall improvement to draggability performance. One possible reason is that since draggability is more difficult to infer visually, the model reached its ceiling earlier.

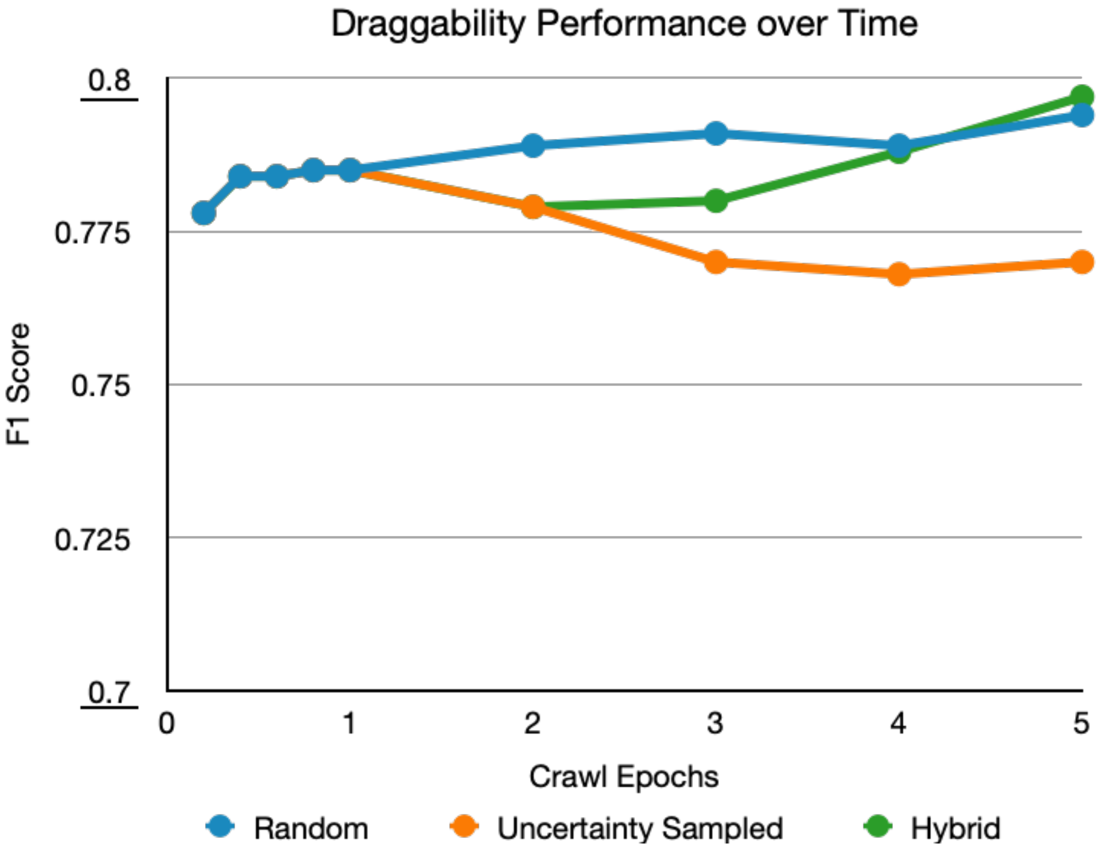


Figure 6.7: Performance of draggability over time. Similar the tappability model, the draggability model improves the most during the first crawl epoch and the rate of improvement plateaus afterward. The hybrid strategy crawler achieves the highest final F1 score of 0.797, and the uncertainty sampled crawler has the lowest F1 score of 0.770.

6.4.3 Screen Similarity

We used our crawler to improve its screen understanding capabilities by using its interactions to validate and retrain the screen similarity model. A more accurate screen similarity model allows our crawler to more reliably determine which app screens it has already visited in an app, and thus increase its exploration efficiency. Screen similarity models have also been used in other types of software engineering applications, such as processing mobile app usage videos [73], automated software testing [198, 200], and automated storyboard generation [66]. Feiz et al. note that due to their labeling technique, their dataset contains more examples of new-screen pairs than same-screen pairs. We mined additional examples of same-screen pairs from our crawler's recorded interactions to augment the original training data and fine-tuned the initial model by lowering the learning rate by a factor of 10. Compared to a baseline condition where the screen similarity model was trained using the unmodified dataset (with the same lowered learning rate), we found that the augmented dataset led to consistently better performance.

Data Generation

We do not introduce a new interaction-based heuristic for collecting labels for screen similarity. Instead, we re-use the data captured from the tappability and draggability heuristics. Both heuristics take two screenshots before initiating an interaction to identify animated or dynamic regions of the screen that could cause false *positives* for tappability and draggability detection. Yet these same examples can also be used to find examples of false *negative* predictions from our screen similarity classifier. We assume that the pre-interaction screenshots belong to the same screen, since any visual variation between them is not caused by a user input. We make similar assumptions about data collected from the draggability heuristic, since the final screenshot is taken before the drag gesture is completed (i.e., before the finger is released from the screen) and is unlikely to result in a new screen. We use these sources to create a dataset of screenshot pairs of same-screen pairs, and we ran our existing screen similarity model to search for incorrect predictions, which can be used to re-train the model. Based on this process, we mined approximately 2000 new examples from each epoch.

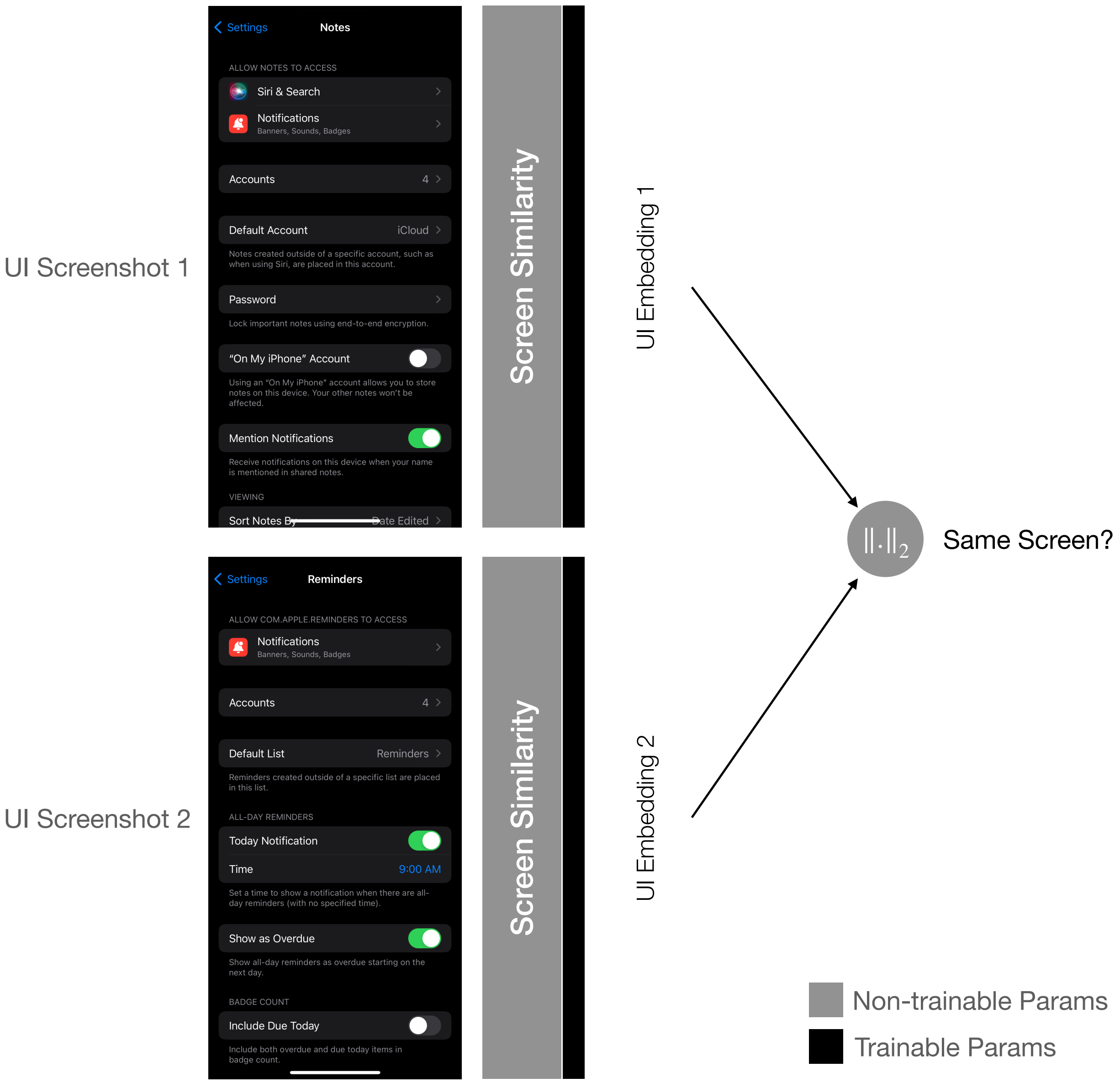
Model Implementation

The screen similarity model was initially trained on a dataset that contained both examples of positive (same-screen) and negative (different-screen) pairs, which made it possible to optimize using a contrastive margin loss [130].

At a high level, the model maps screenshots into an embedding space, and the loss ensures that similar screens are close together (i.e., have a distance less than a margin value) while different screens are further apart.

$$\mathcal{L}_{sim} = \begin{cases} ||\Delta h||_2 & \text{if } s_1 = s_2 \\ \max(0, m - ||\Delta h||_2) & \text{otherwise} \end{cases} \tag{6.1}$$

To fine-tune the model, we use the same training objective but decrease the learning rate to $lr = 1e-5$, which is ten times lower than the original value used to train the model. We initially



Settings — Notes

ALLOW NOTES TO ACCESS

Siri & Search

Notifications
Banners, Sounds, Badges

Accounts 4

Default Account iCloud

Notes created outside of a specific account, such as when using Siri, are placed in this account.

Password

Lock important notes using end-to-end encryption.

"On My iPhone" Account ○

Using an "On My iPhone" account allows you to store notes on this device. Your other notes won't be affected.

Mention Notifications ●

Receive notifications on this device when your name is mentioned in shared notes.

VIEWING

Sort Notes By Date Edited

Settings — Reminders

ALLOW COM.APPLE.REMINDERS TO ACCESS

Notifications
Banners, Sounds, Badges

Accounts 4

Default List Reminders

Reminders created outside of a specific list are placed in this list.

ALL-DAY REMINDERS

Today Notification ●

Time 9:00 AM

Set a time to show a notification when there are all-day reminders (with no specified time).

Show as Overdue ●

Show all-day reminders as overdue starting on the next day.

BADGE COUNT

Include Due Today ○

Include both overdue and due today items in badge count.

Figure 6.8: Architecture of our screen similarity model. The screen similarity determines if two input screenshots are variations of the same UI by i) featurizing each screenshot using a CNN ii) comparing their Euclidean with a threshold value.

tried to use the newly-mined same-screen pairs to fine-tune the model without mixing it with the original dataset. However, this was unsuccessful, since only focusing on the "similarity" term ($s_1 = s_2$) resulted in a failure case where the model learns to map all screenshots to the same point in embedding space, since it is only penalized if similar screens are far away but not if dissimilar screens are close together. Thus, we directly "mixed" in the newly mined examples with the rest of the original dataset, which consisted of 800,000 labeled pairs.

Performance Evaluation

We measured performance with respect to the original dataset's evaluation split because our generated data only contains same-screen pairs, which makes it impossible to compute precision. The results are shown in Figure 6.9. Because the screen similarity model doesn't affect the crawler's selected actions (e.g., attempted taps and drags), we only evaluated our approach on data from the Random crawl. Overall, we found that using the crawler-generated dataset to fine-tune the model led to small but consistent improvements in performance over time. The screen similarity model improved from an initial F1 score of 0.636 to a final F1 score of 0.663. Despite being trained on the original dataset, the baseline model also improved due to the lowered learning rate. A common practice in neural network training is to decrease the learning rate after performance plateaus, which may allow the model to continue improvement. The baseline model improved the initial model to a final F1 score of 0.659.

If the crawler were to run indefinitely, it would need some mechanism to ignore a subset of the collected data to avoid eventual data imbalance due to the collection of only same-screen pairs. Several possible techniques exist for consolidating and distilling datasets to retain the most informative samples [236, 237, 318]. While we believe these methods are applicable, we leave this aspect of validation to future work.

6.5 Discussion

Our experiments revealed that visual UI models could effectively be trained and improved through automated, continual interaction. In this section, we discuss i) the performance of our specific Never-ending UI Learner implementation, ii) other types of interaction-based learning, and iii) the benefits applying these strategies over very long or potentially indefinite period of time

6.5.1 Never-ending UI Learner Performance

In this paper, we conducted a series of experiments that evaluate the Never-ending UI Learner and and its ability to automatically learn UI semantics. Our experiments investigate two key questions: i) what is the best way for an automated crawler to learn about UIs? and ii) how long would it need to run?

Crawling Strategy. Our experiments focused on three crawling strategies for exploring mobile apps: i) randomized crawling, ii) uncertainty sampling, and iii) a hybrid strategy. Overall, the random strategy consistently led to strong performance in all our experiments. We initially

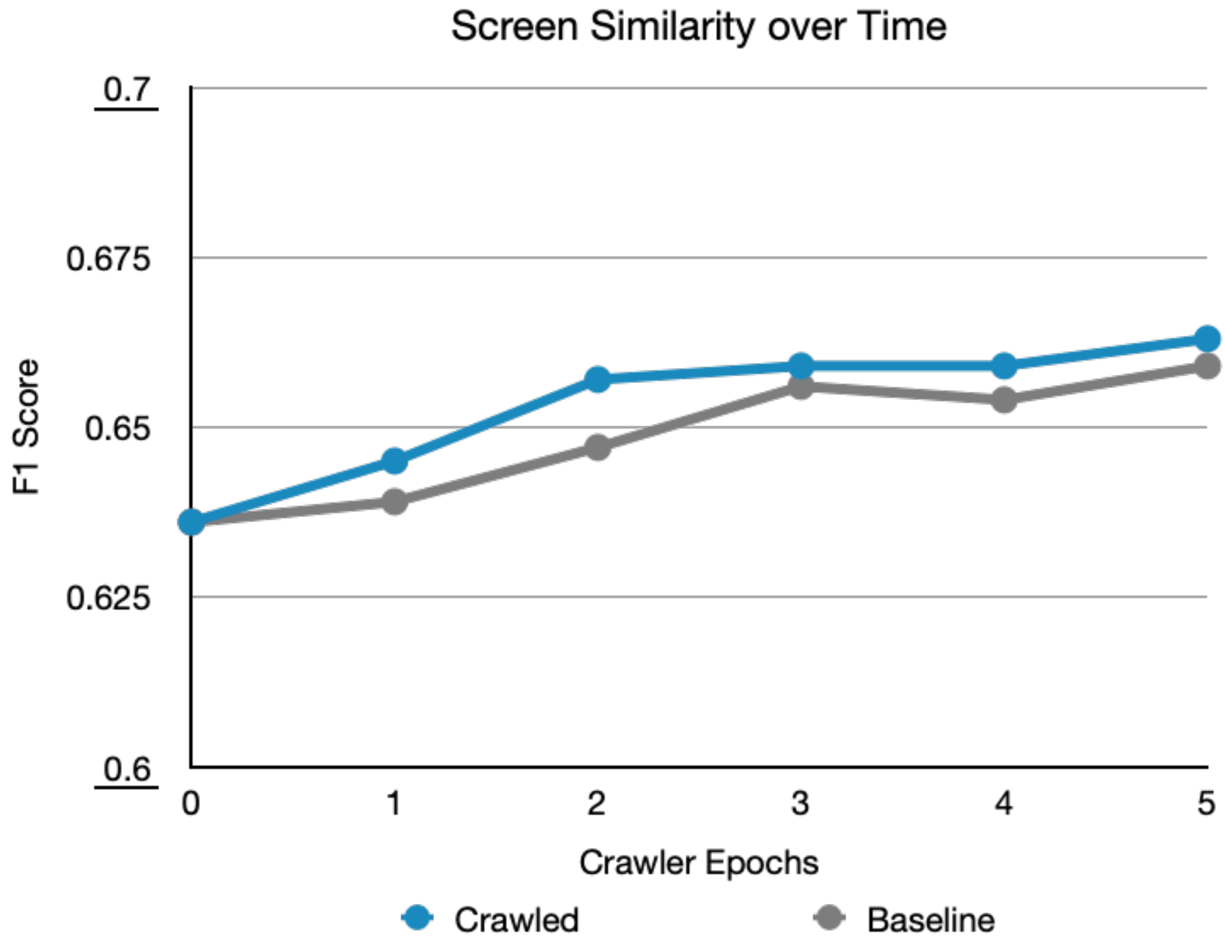


Figure 6.9: Performance of screen similarity over time. We compared i) adding training examples mined from crawls and ii) a baseline of continuing model training on its original dataset with a lower learning rate. The crawler-augmented dataset achieved a final F1 score of 0.663 while the baseline's final F1 score was 0.659.

hypothesized that uncertainty sampling, an active learning technique that improves sampling efficiency by prioritizing examples with low model confidence, would let the model to learn more efficiently and effectively. However, because our crawler updated its models (which are used to compute the prediction confidences) every epoch instead of after each sample (as is often done in applications where uncertainty sampling is employed), it led to imbalanced data collection during subsequent crawls, which decreased performance. The hybrid crawler alternated between random and uncertainty sampling strategies, which allowed it learn from low-confidence predictions while also correcting the distribution shift induced by batched uncertainty sampling. Overall, it led to similar performance to pure random crawling, although it was less consistent. In the draggability task, it initially decreased performance but experience rapid improvement afterward. Ultimately, our experiments do not reveal a clear choice, and we believe there is room for exploring additional strategies [122] and longer-term evaluation, which we leave to future work.

Performance over Time. Even though our crawler is meant to run indefinitely, our experiments focused on a relatively short period of five crawl epochs. Each crawl epoch lasted approximately half a week (clock time) when parallellized across multiple crawler workers and consisted of approximately 500 device-hours of app interaction, data post-processing, and model training. Across all experiments, the Never-ending UI Learner crawled for more than 5,000 device-hours, which was carried out over the span of approximately one month.

Our results show that this window is sufficient to learn accurate models purely from crawler-collected data (tappability and draggability) or fine-tune existing models (screen similarity). Overall, we found that models had rapid early learning followed by slower improvement, which is consistent with empirical observations in machine learning research that suggests an exponential relationship between dataset size and model performance [293]. We believe these small improvements are valuable, since their benefit can be magnified when running over potentially very long periods of time and allow the model to be continuously updated. We plan to continue running the crawler, which doesn't require human supervision, to observe trends over longer periods of time and maximize the potential of our automated learning approach.

Anecdotal Observations. Based on our experimentation, we found several dimensions that affect the performance of never-ending learning systems such as ours. We offer anecdotal observations that may be useful for replication or implementing similar systems.

- *Choosing examples.* In this paper, we primarily explored two methods (random and uncertainty-based) for selecting training examples. We found that random selection is a strong baseline, and active learning approaches (e.g., uncertainty sampling) can be effective with proper hyperparameters. There is more to explore along this dimension, including the use of crawler history to reduce sample redundancy, which we observed in our crawled data.
- *Retraining frequency.* Re-crawling and re-training frequency can affect the system's performance changing the makeup of the training data. The experiments in this paper were run over the span of around one month with an update iteration every 1-2 days, so the app changes that we witnessed were primarily due to changes in dynamic content. We believe that less frequent updates can be effective (e.g., monthly) which could capture more substantial changes such as app updates or newer design guidelines.

- *Evaluation data.* We used a fixed evaluation split to directly compare model performance over time. Using data from the latest crawl may allow for more accurate estimation of real-world performance; however, it is less straightforward to compare models across epochs, since changes may either have been caused by model performance or changes to validation data. Finally, using a dynamic-sized validation set could be useful if the models are deployed in scenarios where performance on both old and new apps are important.

6.5.2 Learning from Interactions

Our work contributes the idea that automated interactions can be used to generate datasets for model-based UI understanding. Most existing datasets use human annotators and crowd workers to produce labels for mobile UI datasets, such as UI element bounding boxes, icon types, and screen similarity pairings. While human labeling has been a de facto standard for creating datasets, especially for domains where the data volume requirements for self-supervised learning are not feasible, crowd worker-generated annotations are known to be susceptible to errors and biases [72, 253]. Furthermore, many tasks implicitly encode a degree of subjectivity. One such example is tappability prediction, where annotators use their own judgment to decide whether a particular UI element in a screenshot is tappable. Labels for such tasks are known to be noisy in practice, and are often averaged or voted on from multiple crowd workers, further increasing the time and cost of human-produced labels [278]. In contrast, automated interaction-based learning can significantly mitigate annotator biases, since labels are produced through hypothesis testing. However, there may be cases where encoding perceptual information into labels can be useful, such as giving feedback to designers on *perceived* tappability. In other cases, such as generating accessibility information, labels more closely aligned with ground truth may be preferred. Understanding the trade-offs between these methods and their impact on model alignment is an opportunity for future work.

In our work, we showed that interaction-based learning can be used to model element (tappability), container (draggability), and screen-level (screen similarity) semantics in mobile UIs. For our tested applications, we found that heuristics that operated with knowledge of the entire interaction made label generation relatively straightforward. However, highly accurate heuristics did not always lead to highly accurate models since the model had to make the same prediction with access to less data (i.e., only visual information from a static screenshot). Some types of semantics were more conducive to visual modeling than others. Our tappability model achieved high classification performance, with an F1 score of 0.860. On the other hand, draggability was much harder to predict from a screenshot (F1=0.797).

A natural question to explore is: what other types of semantics can be learned through interaction? For example, related semantics such as "press-and-hold" functionality can be discovered, and textboxes can be better understood by observing what kind of software keyboard (e.g., email or numeric keyboard) appears when it is tapped on. Could this approach be extended to the problem of UI element detection more generally, which currently relies heavily on human annotation? There are many details that would need to inferred, such as the size and shape of UI elements, and of course the element type. Many more interactions would be needed from the crawler to determine a bounding box for a given element, and it might be difficult to infer complex element types, but a working system that could do this might be able to learn about custom controls and other non-standard elements that current models cannot deal with today. Better automated understanding of UIs can not only benefit downstream applications directly, but also collect better data to train models.

6.5.3 Benefits of Never-ending Learning

Our crawler is meant to be run indefinitely, allowing it to accumulate examples and train over long periods of time. In our paper, we experimented with several variables (e.g., training hyperparameters and exploration strategies), which was only feasible by focusing on a relatively short period of time for each condition (5 crawl epochs). Even from this short time-span, we could train models for UI semantics "from scratch" and observed consistent improvements to performance afterwards, but we believe that our models are yet to reach their maximum potential. In addition to its performance benefits, never-ending learning allows machines to learn from diverse sources of data. Never-ending learning can help machines identify and learn from mistakes, especially those caused by shifts in data distribution caused by trends in app usage and design trends.

Never-ending learning also introduces new challenges, like "catastrophic forgetting," the possibility of erasing previously learned information by training on new data, and difficulties associated with large, ever-growing datasets. In this paper, we conduct a preliminary exploration of methods to address some of these challenges, such as uncertainty sampling, which can help prioritize certain types of data. Our literature review uncovered many other possible machine learning techniques that involve training the model training process [56, 161, 202, 263] or distilling the collected dataset relevant samples [236, 237, 318]. We expect that they will be useful for scaling and maximizing the performance of never-ending UI learning.

6.6 Limitations & Future Work

Our current implementation of a Never-ending UI learner is limited and presents opportunities for future exploration.

First, our current crawler is implemented using a specific set of tools and infrastructure customized for our target platform (iOS). While we did not run experiments on other types of UIs (e.g., Android, web-based interfaces), we expect our results to be generalizable, since our approach does not rely on any platform-specific metadata or APIs, and previous research has shown semantic overlap between mobile and web UIs [329]. Our experiments primarily focused on free apps that did not require authentication (e.g., registering and making an account), which biased the set of UI screens reached by crawling. We used manually-designed and verified heuristics for a small set of semantics for tappability and draggability. We believe that many other aspects of UIs and interaction can be formulated using similar methods. Another limitation of our current experiments is that we did not investigate the effect of different randomized train/test splits, which could provide additional insight into the robustness of our method. Because experiments took roughly a month to complete, the time and compute costs for repeated trials would have been prohibitively high. However, since our list of apps is sufficiently large and randomly shuffled, we do not expect large variations in performance across different randomized splits.

Personalized interaction traces collected over long periods of usage can improve the performance of models for rarer, niche apps, although a privacy-preserving approach would be needed (e.g., on-device training). An alternative direction is to allow our crawler to automatically learn interaction sequences to discover and label new aspects of UIs, instead of executing manually-defined heuristics. We expect future versions of our crawler to incorporate techniques from related machine learning fields, such as reinforcement learning.

Finally, our crawler could benefit improved UI understanding capabilities. First, our crawler's primary representation of screens that it visits is a list of UI elements, which are used to navigate and discover other parts of the app. A more effective way of representing screens could lead to more efficient crawling [325]. For example, since properties of list items are similar, the crawler could reduce unneeded interactions by tapping on one list item and propagating the label to others. Icon semantics [62, 64, 213] are also helpful for inferring the result of certain interactions. For example, tapping on a "camera" icon may open the system camera app, which would disrupt the crawl. Since the goal of the crawler itself is to train such UI models, we believe that integrating these additional models into the never-ending learning framework is a natural next step.

6.7 Conclusion

In this work, we presented a technique for continuous extraction and modeling of user interface semantics through interactions, which we refer to as "never-ending learning of UIs." We implemented a mobile app crawler that downloads, installs, and crawls thousands of apps to observe UI semantics and affordances in real-world apps, and we use interaction-based heuristics to generate large datasets for training three types of UI understanding models i) tappability, ii) draggability, and iii) screen similarity. We found that models trained in this way can be more accurate than those trained from human-annotated screenshots and continue to improve with access to more training examples. The highly automated nature of our approach allows us to apply it indefinitely, with little to no human supervision, which can maximize their performance and utility to downstream applications.

7

Reflow: Automatically Improving Touch Interactions in Mobile Applications through Pixel-based Refinements

7.1 Preamble

Content in this chapter is based on a previously published technical report and workshop paper at the CHI 2022 Computational UI workshop [327].

This chapter presents Reflow, a system that imagines a future where any existing app can be dynamically optimized based on user-specific and other in-situ factors. Because many apps are not constructed in a way that integrate with OS features [346] and do not expose application semantics, previously described approaches (Chapter 3) cannot work in many circumstances. Instead, we propose an "end-to-end" that requires only the pixels of an existing input UI and produces the pixels of an optimized one. Reflow combines the output of pixel-based semantic prediction models with a user-specific model generated from a calibration task.

Touch is the primary way that users interact with smartphones. However, building mobile user interfaces where touch interactions work well for all users is a difficult problem, because users have different abilities and preferences. We propose a system, Reflow, which automatically applies small, personalized UI adaptations, called *refinements*—to mobile app screens to improve touch efficiency. Reflow uses a pixel-based strategy to work with existing applications, and improves touch efficiency while minimally disrupting the design intent of the original application. Our system optimizes a UI by *(i)* extracting its layout from its screenshot, *(ii)* refining its layout, and *(iii)* re-rendering the UI to reflect these modifications. We conducted a user study with 10 participants and a heuristic evaluation with 6 experts and found that applications optimized by Reflow led to, on average, 9% faster selection time with minimal layout disruption. The results

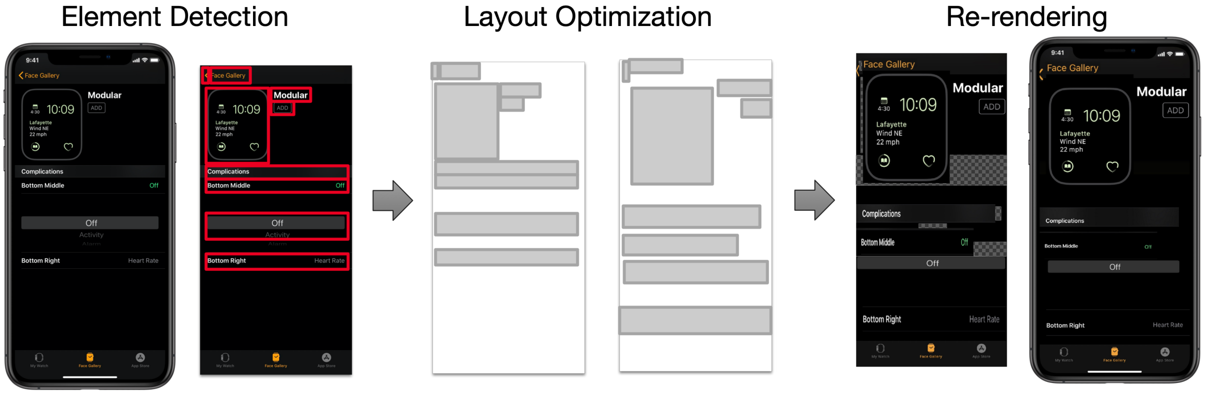


Figure 7.1: Reflow *(i)* detects UI elements from pixels, *(ii)* optimizes the UI layout for a personalized difficulty model, and then *(iii)* re-renders the visual UI with the new layout. In this example, the UI elements are correctly detected, and the new layout includes larger buttons that are more spread apart. Simply moving and stretching the UI elements causes gaps and distortions, which Reflow fixes using additional post-processing methods.

demonstrate that Reflow's refinements useful UI adaptations to improve touch interactions.

7.2 Introduction

Touch is an ubiquitous way of interacting with smartphones. However, building mobile user interfaces (UI) where touch interactions work well for all users is a difficult problem, because users have different motor abilities, skills, and even preferences [322]. For example, consider a right-handed user who wants to access a menu item on the left side of the screen. For larger screens, this menu item is more difficult for the user to touch with their right hand than for a left-handed user.

UI adaptivity is a promising approach towards improving touch interactions, because it allows systems to dynamically personalize the UI and tailor the UI to the users' needs. But for UI adaptivity to be practically useful for real-world apps, it must support two goals. First, the technique should be generally useful across a broad range of existing mobile applications. Second, the technique should apply adaptations in a way that respects the design intentions of the original applications. In other words, we expect that drastic UI adaptations are likely to make the user interface less familiar to the user and disruptive to the overall user experience [46].

To operationalize these goals, we built a system, *Reflow*, which automatically applies small UI adaptations—called *refinements*—to mobile existing app screens to improve touch efficiency. Towards the first goal of supporting broad applicability, Reflow is entirely pixel-based: the system does not need knowledge of the applications' dependencies or view hierarchy to make its UI adaptations. Towards the second goal of respecting design intent and minimally disrupting the user experience, Reflow incorporates the theory of microstrategies in its model [93, 96, 124]. Microstrategies suggest that even small, principled adaptations to the user interface can significantly improve task efficiency—particularly over cumulative usage—and we postulate that the same principle applies when personalizing touch-based mobile applications.

Reflow supports personalized optimization by constructing a spatial map from usage data, which identifies difficult-to-access areas of the screen (*e.g.,* elements on edges of the screen requires users to reach and reposition their hand to select). Reflow then *(i)* automatically detects the UI elements contained on the screen, *(ii)* uses a machine learning model to optimize the UI layout to better support the difficulty map, and then *(iii)* re-renders the existing UI pixels to match the new layout (Figure 7.1). Reflow improves on existing approaches because it works with a range of existing mobile applications and enables an end-to-end pipeline from layout optimization to re-rendering the application screens.

To evaluate Reflow, we first conducted a study with 10 participants, where we found it improved interaction speed by 9% on average, and improved interaction speeds by up to 17% for some UIs. From lessons learned, we made further improvements to this model by detecting and applying an additional set of UI constraints (relative positioning, alignment). We then conducted a heuristic evaluation based heuristics for evaluating UI layouts [284, 306] with 3 accessibility and 3 design experts to validate if these improvements make acceptable trade-offs between touch efficiency and layout preservation. Feedback from our expert evaluators indicated that refinements were likely to improve selection time while avoiding significant disruption to the UI. The results of this work demonstrate that refinements are a useful UI adaptation technique to improve task efficiency for touch interactions.

The contributions of our paper are as follows:

- We propose an approach based on the theory of microstrategies [124] for improving touch interactions through *refinements*, which are small modifications to the original UI. Based on this approach, we present Reflow, an end-to-end system for personalizing any existing mobile app using only its pixels. We describe our implementation in several modular steps: *(i)* element detection, *(ii)* layout refinement, *(iii)* UI re-rendering, which may be applied to other UI adaptation systems.
- We conduct two evaluations that provide evidence for the effectiveness of Reflow's automatic UI refinements. From our user study ($n = 10$), we find that the refinements automatically applied by Reflow result in more efficient touch interaction (average speedup of 9%, up to 32%). Furthermore, we conduct a heuristic evaluation with 3 accessibility and 3 design experts, validating that the changes induced by Reflow are likely to improve selection time while being minimally disruptive. Qualitative feedback from our expert evaluators provide additional rationale for the acceptability of these trade-offs.

7.3 Example Usage Scenario for Reflow

To motivate how Reflow can be used to improve touch interactions on mobile apps, we provide an expected usage scenario where Reflow would be enabled globally through the mobile operating system.

Scenario: Alice has recently purchased a new smartphone. Alice's new smartphone is able to install and run all of her favorite apps, but the device is slightly larger than her old smartphone. Because of this, she can no longer comfortably reach UI controls on the far edges of the screen while using the device with one hand. While she is still able to use her favorite apps, Alice finds

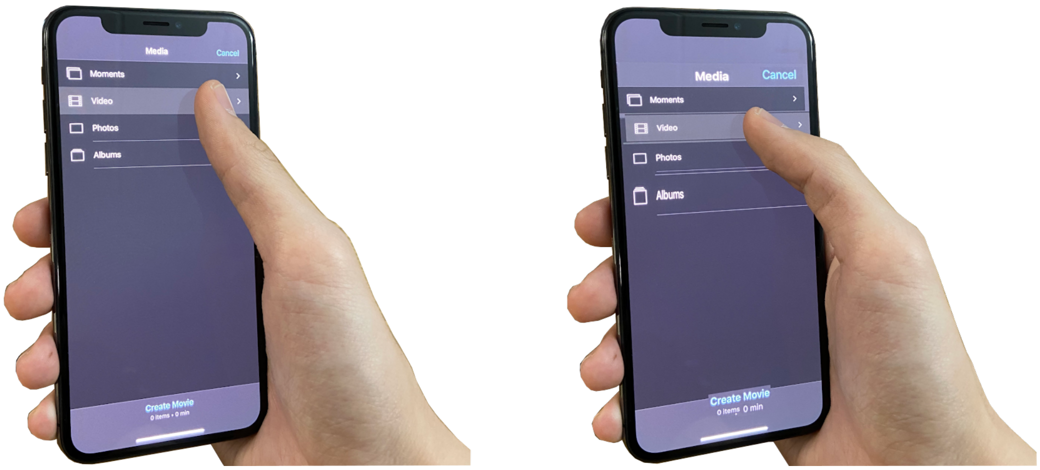


Figure 7.2: Reflow optimizes existing third party apps using only UI information from pixels and a spatial difficulty map corresponding to a user's abilities. An app screen with short menu items near the top of the screen is difficult to use (Left). Reflow automatically optimizes the layout of on-screen elements, making each menu item taller and shifting items down (Right).

her routine interactions with the apps inconvenient.

The Reachability accessibility feature built into iOS[1] addresses the challenge of touching items in the upper half of the screen by enabling users to swipe down on the lower portion of the screen to move the upper half of the screen down to the lower half. While this is useful to many people, it requires an additional touch interaction (swipe down) which slows Alice down and feels like more support than she requires.

Setup: Alice opens the settings application on her smartphone and turns on the "Reflow" toggle button. If this is the first time the setting has been enabled, Alice is brought to a setup screen that that initializes the Reflow system. This screen is similar to the first-time setup process of some biometric authentication features (*e.g.,* Face ID). The setup screen asks Alice to perform a calibration task, and measures how quickly she can select targets located at different parts of the screen. With this usage information, Reflow creates a personalized profile for her that identifies difficult-to-reach areas on the screen.

Usage: Once enabled, Reflow automatically intercepts and applies refinements to app screens before they are displayed to the user, a form of *manifest interface* [87]. The UI produced by Reflow is interactive, as it automatically redirects input events to the original app screen, using techniques similar to previous work [292, 345]. Alice notices that the location and size of UI elements on apps have only slightly changed, and that the overall appearance and structure of app UIs are still very similar (Figure 7.2). Because of this, she is once again able to comfortably use her favorite apps, as she was previously able to do with her old phone, but she does not need to re-familiarize herself with the refined UIs.

Customization: Alice may decide to customize the behavior of Reflow by opening its settings panel. In this panel, she may define lists of apps that she wishes the refinement feature

[1] https://support.apple.com/guide/iphone/touch-iph77bcdd132/14.0/ios/14.0#iph145eba8e9

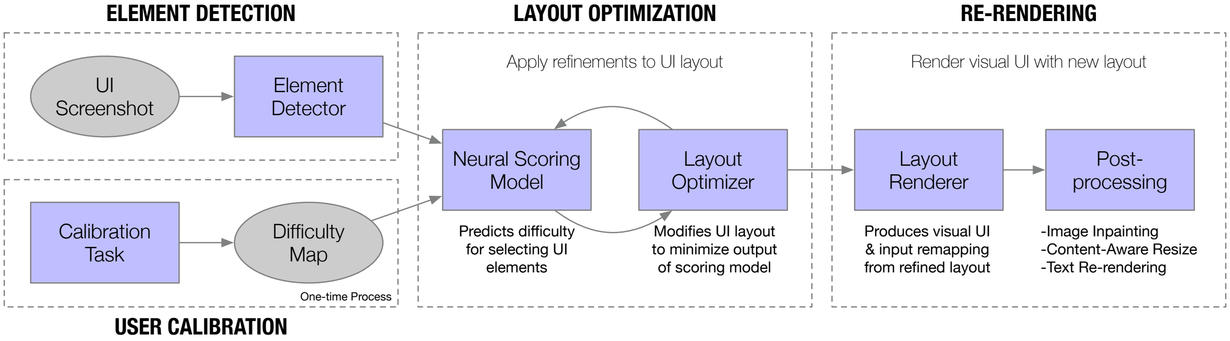


Figure 7.3: A block diagram describing the architecture of Reflow. Reflow consists of 4 main stages: *(i)* user calibration, *(ii)* element detection, *(iii)* layout optimization, and *(iv)* re-rendering. Together, these stages allow an existing UI screenshot to be adapted to a personalized difficulty model.

to include or exclude. This could be useful for disabling the feature on apps that Alice already finds easy-to-use on her new smartphone. Similarly, Alice can define a shortcut (*e.g.,* gesture or detected activity) to quickly toggle Reflow's functionality based on when it is useful. Finally, the settings panel allows her to reset or re-calibrate the feature to reflect updated preferences or physical affordances (for example, a smartphone case that makes it easier to grip the device).

7.4 Reflow

Reflow is an end-to-end system that produces a refined UI from an original app's screenshot. Reflow's operations occur in several stages (Figure 7.3). First, the user is asked to perform a calibration task, which is used by Reflow to construct a difficulty map characterizing areas of the screen which may be hard-to-reach. During runtime, an element detector is used to extract the layout of an existing app screen. This layout is then optimized by repeatedly *(i)* predicting the selection time for UI elements given the UI layout and user-specific difficulty map and *(ii)* modifying the layout to minimize this predicted value while respecting stopping conditions designed to detect large disruptions (*e.g.,* overlapping, other constraints). Finally, the refined layout is re-rendered into a refined graphical user interface, where it can be presented to the user and made interactive.

7.4.1 User Calibration

Calibration Task

To characterize the areas of the screen that are hard-to-reach for a user, we designed a one-handed calibration task that a user performs when first enabling the feature. We focused on one-handed touch interactions because these motor activities are more likely to cause selection challenges in user interfaces, particularly given the limited range and flexibility of the thumb [39, 58, 235]. Users were asked to select on-screen targets, and the system recorded the *(i)* selection time and

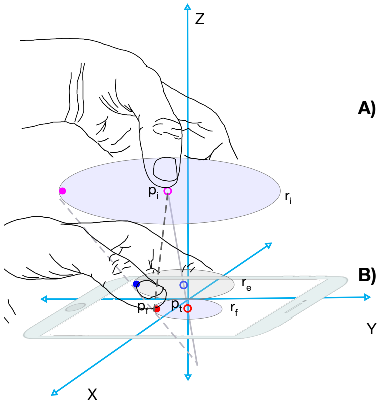


Figure 7.4: A user taps a location on the screen by lowering a finger from a starting position p_i (A) to a target location on the screen p_t (B), whose path is shown as a dark gray dotted line. The circles represent region on the screen where the finger is expected to land given the current position. As the finger descends towards the screen, the circle shrinks, as the user "hones in" on the target location. The user's finger makes contact with the screen at p_f, which is a distance r_f away from p_t (B).

(ii) selection error (*i.e.,* difference between the target position and recorded tap position). These targets were uniformly spaced in a 4x8 grid, with a single, randomly selected target highlighted at a time. After a target was selected, the next one was not immediately displayed; instead, it was delayed for the remainder of a timeout value. This was done to allow the finger to return to a "rest position" after selection and reduce the influence of the previous target's location on the finger's starting position. We set the timeout value to 3 seconds, based on our early observations and estimation of how long this process would take.

We conducted a data collection study with 10 participants (7M/2F/1 prefer not to disclose, ages 22-40, recruited within our organization) to characterize the input error for users and to initialize our system. We performed data collection remotely using video conference software. Participants were asked to install an app on an iPhone, which was needed to run our software. 4/10 of our participants used an iPhone Xs, 2/10 used an iPhone 11, 3/10 used an iPhone Xs Max, and 1/10 used an iPhone 11 Pro. Users were asked to hold their device with one hand and tap on targets using the same hand that they were holding the device with.

During the study, participants were asked to select targets placed at different locations on the screen. In total, the study required less than 30 minutes and included both a practice and evaluation session. 9/10 participants held the phone in their right hand.

Difficulty Map

Using calibration data from each user, we generated a personalized difficulty map [232] that estimates the relative difficulty of accessing any on-screen location (including locations between two of the original calibration points).

We calculated difficulty by combining two measurements from the calibration data: *(i)* input

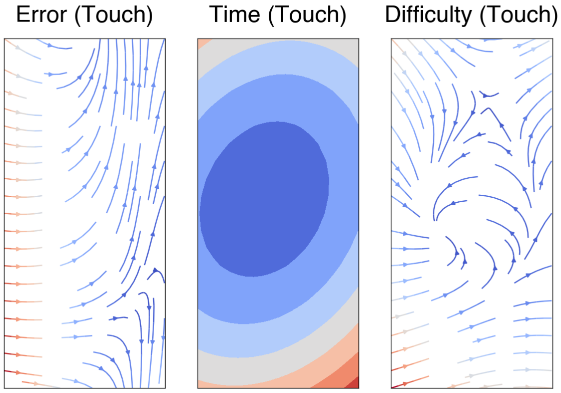


Figure 7.5: An example of a difficulty map generated by our user calibration process. Data from the calibration task is first used to compute input error (Left) and selection time (Center) independently. We combine these two measurements to produce the "adjusted error", or difficulty map (Right).

error and *(ii)* selection time. We introduce a procedure to "normalize" the raw error measurements by selection time. Our procedure is based on the intuition that if a user selects two targets with equal accuracy (*i.e.,* equal input error), the one that took longer to select (*i.e.*, higher selection time) is more difficult. We refer to this computed value as the "adjusted error", and it represents an estimation of input error given a constant selection time. In other words, if a user was only given one second to select anywhere on the screen, what is the error we would expect for a given location? Our approach is informed by prior work which suggests that selection-based interactions experience a tradeoff between speed and accuracy [219].

Our computation of adjusted error is based on the standard Fitts's law equation. Figure 7.4 shows a user's finger, initially located at p_i (*i.e.,* resting position), selecting a target located at p_t, which is located a distance A away.

$$\begin{aligned} t &= a + b \cdot \log_2(A) \\ t &= a + b \cdot \log_2(||p_f - p_i||) \end{aligned} \tag{7.1}$$

If the resting position p_i is assumed to be constant for a single user (which previous research has shown is plausible [176, 223]), we can estimate its location as a model parameter. To do this—in addition to the standard Fitts' model parameters a and b—we also learned p_i by fitting Equation 7.1 to a dataset of pairs $\langle t_i, p_f \rangle$ using non-linear ordinary least squares. The initial value of p_i was set based on the resting location area identified in previous work [176], and we constrained the position of p_i so that it lies at most 5 inches above the device screen (a reasonable upper bound for the thumb's distance to the screen when holding a smartphone) and within the screen's x-y bounds.

To estimate the error magnitude ϵ at a constant time t_n, we solve the following system of equations, where a and b are from the Fitts's model, and $\mathbf{d}$ is the distance the finger has traveled at t_n.

$$((A, r_f, 1) \times (0, r_i, 1))(\mathbf{d}, \epsilon, 1)^{\mathbf{T}} = 0 \tag{7.2}$$

$$\begin{aligned} \mathbf{d} &= 2^{\frac{t_n - a}{b}} \\ \epsilon &= \frac{(r_f - r_i) \cdot \mathbf{d}}{A} + r_i \end{aligned} \tag{7.3}$$

Using this procedure, we compute the adjusted error at each of the original calibration points (4x8 grid), then fit a bivariate polynomial (similar to a 2-D spline) to interpolate values in between.

To summarize, the output of the user calibration step is a function that returns the adjusted error for any location on the screen. The adjusted error at location (x, y) is an estimate of the offset of where a user's touch would land if given t_n seconds to tap a target located at (x, y).

7.4.2 Element Detection

In this stage, Reflow extracts the locations of on-screen UI elements using a CNN-based object detector and grouping heuristics [347]. We performed additional post-processing on the output of the object detector to further improve performance. First, we removed any detection that overlaps (by keeping the one with the higher confidence) or contains another (if a detection contains a text element, then the container is removed; otherwise, the children are removed). The resulting layout contains no overlapping elements. Because the downstream re-rendering stage (7.4.4) involves cropping and moving image patches, the quality of element bounding boxes identified in element detection have affect the quality of the final output. We employed a heuristic that refines the positions of bounding box edges by repeatedly *(i)* computing the mean color of the pixels of each edge, *(ii)* selecting the edge whose mean color is most dissimilar from the others, *(iii)* adjusting its value by a small increment in the direction of improvement.

The output of the element detector is the UI layout of the original screenshot: $\Theta = \{\theta_1, \theta_2, \cdots, \theta_n\}$ where $\theta_i = [x, y, w, h]$ describes the location and size of the i-th bounding box.

7.4.3 Layout Optimization

Neural Scoring Model

Using a difficulty map, we estimated the error a user would experience when selecting individual elements of the UI. For a given app layout we define a scoring function $S(\Theta)$ which quantifies how likely the user is able to successfully select each element in the screen. We define the scoring function as

$$S(\Theta) = \sum_{\theta \in \Theta} \iint\limits_{R_\theta} P_\theta \, dA \tag{7.4}$$

where P_θ is the predicted distribution of where the user's finger will actually land when attempting to click the middle of an element θ. To estimate P_θ, we centered 2-D Gaussian on the center of UI element θ and set its covariance based on the adjusted error at that location. We compute $S(\Theta)$ using Monte Carlo integration. For each UI element, n samples are drawn from the 2-D Gaussian parameterized by the adjusted error at its center. We set $n = 30$ based on empirical observations of how many samples were needed for consistent results. We scored a screen by counting the number of true positives over total number of points.

To further improve upon the speed and efficiency of our UI scoring function, we used a neural network to learn the result Monte Carlo scoring function. Our model architecture (7.6)

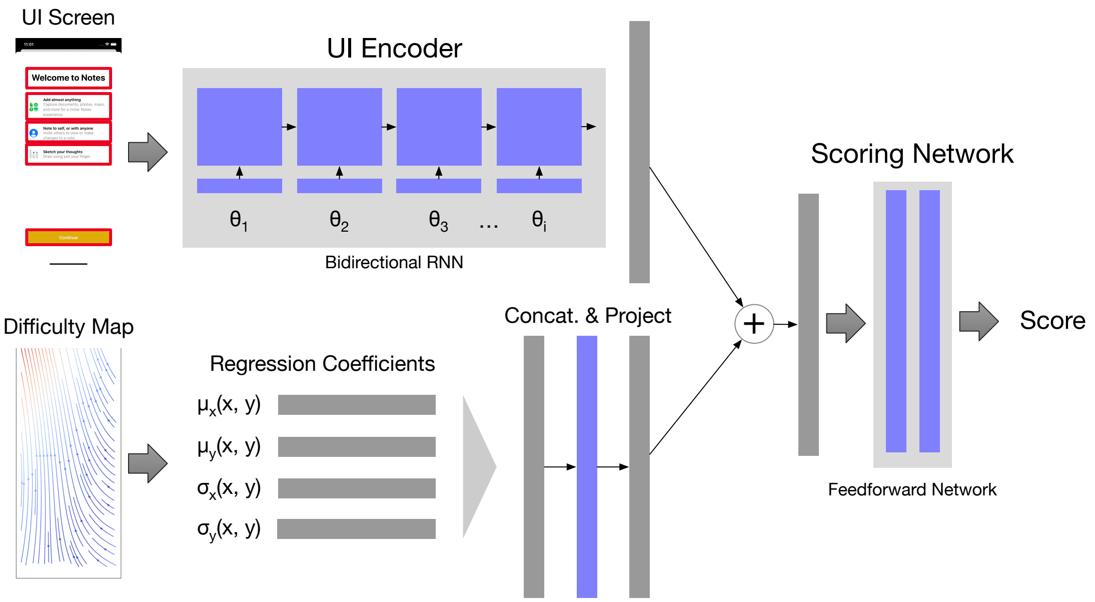


Figure 7.6: The architecture for the neural scoring model used for scoring a UI screen given a spatial map of error. The network encodes the layout of UI elements using a bidirectional RNN and encodes the spatial difficulty map using the coefficients of a 2-D polynomial function fitted to the calibration points. These encoded representations are combined and fed into a feedforward network.

consists of an LSTM-based UI layout encoder [90, 194] and a difficulty map encoder. Because we parameterize the difficulty map as a model input, it allows for personalization at runtime—allowing re-calibration and removing the need to retrain the network for every new user.

We trained our neural scoring network on two datasets. The first was a large annotated dataset of 77,000 iOS app screens introduced in prior work [347]. The second dataset consisted of spatial difficulty maps collected from our user calibration dataset. Because the second dataset is relatively small, we augmented it by introducing perturbations and randomly generated difficulty maps. The network was trained by randomly selecting an application screen and a spatial difficulty map and computing the score using our Monte Carlo algorithm, which was used as the ground truth. We trained our network using stochastic gradient descent until the validation loss stopped improving.

Layout Optimization

To optimize a UI layout, we use predictions from our neural scoring model to make modifications that reduce expected interaction difficulty. One benefit of our model is that it is differentiable (since it is a neural network) and thus, given an initial screen layout and its corresponding score, we can use gradient-based optimizer [160] to find an improved layout with a better score. In service of our refinement approach to UI adaptation, we guide optimization and stop the process if disruptive changes are detected.

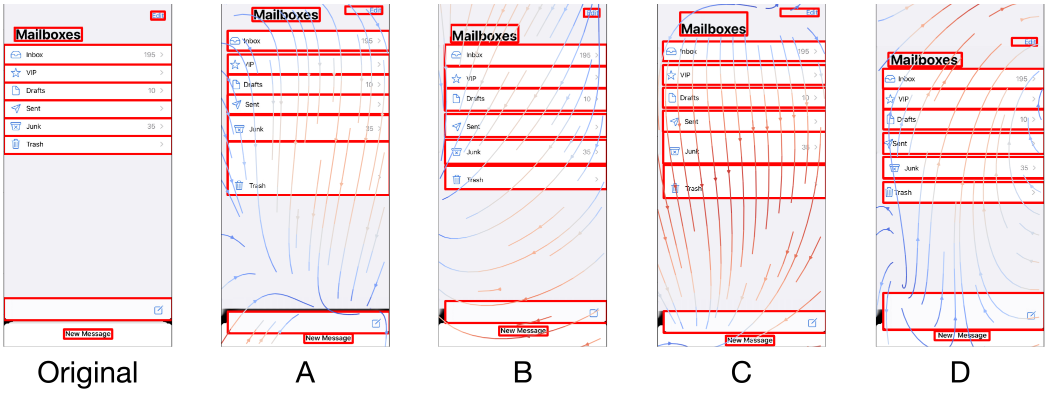


Figure 7.7: An email app ("Original") is optimized using four different spatial difficulty maps ("A", "B", "C", "D"). The result of the optimization process moves UI elements away from regions of higher relative difficulty (red) to lower relative difficulty (blue). Depending on the direction of error where an element is located, it may be stretched horizontally or vertically.

First, we add a regularization term that penalizes layouts that are proportionally dissimilar to the original. This regularization term is defined as the cosine distance (D_C) between the pairwise L_1 displacements of each UI element ($\phi(\Theta)$). This does not penalize the layout for increasing in size but attempts to maintains relationships between neighboring UI elements.

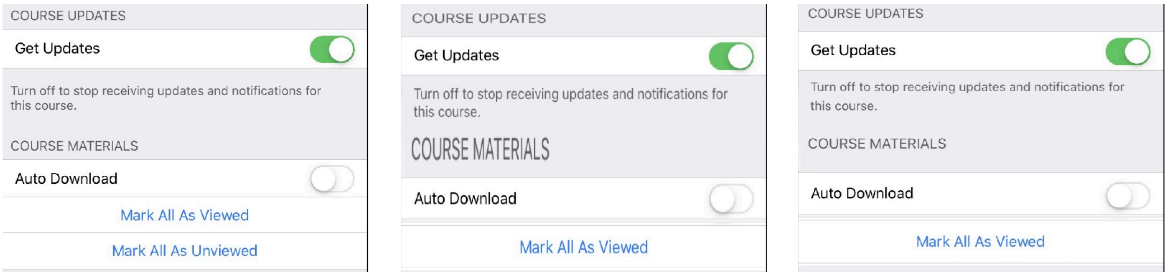
Next, we add "corrective procedures" after each optimization step to guide the optimization of certain properties (*e.g.,* size and element). We clamped element parameters to ensure they stay within a certain range and to prevent elements from becoming too small or large. We also use a routine that detects overlapping regions and resolves them by moving overlapped elements further apart. The overlap removal algorithm repeatedly tries to find intersecting region between pairs of UI elements, and if one exists, shifts them apart in the axis of least overlap. This process is repeated until no more overlaps are detected or a max number of iterations is reached. Despite our precaution, the final UI may still appear to contain overlaps due to inaccurate element detection (*e.g.,* detection bounding box does not fully enclose element or includes multiple elements) and artifacts introduced by our re-rendering process.

Finally, we implemented an early stopping condition that is triggered when the overlap removal algorithm detects overlaps that are unresolvable, that is, overlaps that cannot be resolved by moving elements further apart.

Figure 7.7 shows examples of our algorithm's output for an email app optimized using different difficulty maps.

7.4.4 Re-rendering

Using the refined UI layout produced by layout optimization, we produce artifacts that are needed for end-user interaction: *(i)* visual representation of the UI (UI screenshot) and *(ii)* mapping between the original and refined UIs, needed for input redirection [292, 345].



Original

Resized (Scaling)

Resized (Content-Aware)

Figure 7.8: An example of how content-aware resizing (Right) improves upon standard scaling (Center). Compared to the original screenshot (Left), scaling (Center) introduces distortions that make text less legible. Our approach (Right) preserves textual content.

Layout Renderer

We use the refined UI layout and re-render the UI back into a visual representation. First, Reflow outputs a mapping between regions of the original UI and the refined UI. This mapping can be used to update the interactive regions of the UI using input/output redirection methods [292, 345] (*e.g.,* clickable bounding box a button is updated to reflect its new optimized position). To align the screen's visual appearance with the updated regions, image patches from the original screen are translated and resized to their new locations.

Post-processing

We use several post-processing strategies to preserve relevant aesthetic qualities (*i.e.,* legibility of text). We briefly describe three such techniques we used, which improve the legibility of: *(i)* background areas, *(ii)* text, and *(iii)* image content.

Image Inpainting. Moving and resizing elements can result in "holes" so we employ an inpainting technique to generate visually plausible replacements. We experimented with many different inpainting algorithms [37, 41, 302], and found that most methods produced results of similar visual quality, since inpainted regions are unlikely to contain complex textures or structural features (most visual content is contained inside of the UI elements themselves). Our implementation uses flow-based inpainting algorithm [41] included in the OpenCV library [49].

Content-Aware Resizing. Because standard rescaling methods can distort elements (7.8), our system uses an optimized method for resizing these regions. We first create an image canvas with the target region's size. The source image patch resized (retaining the original aspect ratio) to maximum size fits within the target dimensions. Leftover space is filled using image inpainting.

Text Re-rendering. We explored a method specifically for resizing and replacing text (*e.g.,* translation example application). Our approach to resizing text involves detecting and extracting text using optical character recognition (OCR), re-rendering the text with the correct background and foreground colors, then inserting the result at the target location. We used an off-the-shelf OCR system that recognizes text from images [289]. We estimated the original text's font size by rendering it using a known font and comparing its dimensions to the size of the original image patch. Background and foreground color are extracted from the original image patch by performing k-means clustering on the pixels and extracting *(i)* the largest cluster (background color), and *(ii)* the cluster which is furthest away from the background color in pixel space (foreground color). We created an image patch corresponding to the target dimensions and rendered the text with the correct foreground and background colors.

7.4.5 Prototype Implementation

For the purposes of our prototype, the Reflow system was implemented on a remote server. Screenshot images are sent from the iOS device to the remote server, which returns a re-rendered screenshot image that contained the UI refinements. For purposes of the user study (described next), the re-rendered screenshot is displayed to the user and touch events are recorded, which allowed us to conduct our user study. Prior work has demonstrated how such a re-rendered graphical UI could be used in a more advanced proxy setup to control the original underlying mobile app, which is how we imagine it would be used with existing applications in practice. We leave that implementation, and some difficult details (*e.g.*, handling scrolling GUIs) to future work.

7.5 User Study

7.5.1 Procedure

We empirically evaluated the performance of our system through a 45-minute user study. We recruited 10 participants (5M/4F/1 Prefer not disclose, ages 24-36) within our organization. Similar to how we collected data in the calibration task (7.4.1), we conducted our usability study remotely using a specialized data collection app and video conferencing software. The breakdown of devices used by our participants were: 1/10 iPhone X, 5/10 iPhone Xs, 1/10 iPhone XR, 1/10 iPhone Xs Max, 1/10 iPhone 11, 1/10 iPhone 11 Pro. 8/10 participants held the phone using their right hand.

The usability study consisted of two phases, a calibration phase and a navigation phase, which included a practice session. During calibration participants performed the user calibration task, which was used to generate personalized difficulty maps. Participants were then given a short 5 minute break. During the navigation phase, participants were presented with a series of app screens with a target UI element that was highlighted. Participants were instructed to select the target element, which brought them to the next screen.

We chose 3 apps where users had to navigate through a total of 5 screens each (Figure 7.9). 3 apps were chosen to constrain the study design to fit under our time constraints, and we selected default applications on iOS that users are likely to be familiar with, since they come pre-installed.

Table 7.1: Navigation Time Results from our User Study

	Screen		**App**	
	Original	Reflow	Original	Reflow
Books	1.1 ± 0.4	1.0 ± 0.3	4.2 ± 1.0	3.9 ± 0.9
Clock	1.1 ± 0.6	1.0 ± 0.5	4.4 ± 1.5	3.8 ± 1.3
Photos	1.2 ± 0.5	1.0 ± 0.4	4.7 ± 1.4	4.1 ± 1.1
Overall	1.1 ± 0.5	1.0 ± 0.4	4.4 ± 1.3	4.0 ± 1.1

Navigation times from our user study. Values shown are the times ($M \pm SD$) needed to navigate a single screen. We report results for both per-screen and per-app navigation. Screen navigation refers to the time taken to advance one screen, while app navigation refers to the time taken to complete all screens for the app. For both measurements, Reflow's refinements resulted in 9% faster navigation, on average.

All of these apps and screens were outside of the training set used to train our scoring model. For each of the screens, we used our system (running on a remote server) to optimize its layout using the user's difficulty map generated during the calibration phase.

7.5.2 Results

Table 7.1 shows the results of our user study. We report both screen (time taken to navigate a single screen) and app (time taken to navigate all screens in an app) interaction times. For both, Reflow's refinments resulted in an average of 9% faster navigation. This speed-up was significant for screen-level navigation ($T = 2.76, p < 0.01$, Cohen's $d = 0.23$) and approached significance for app-level navigation ($T = 1.99$, $p = 0.06$, Cohen's $d = 0.38$), due to small sample size for apps. The small–medium effect size suggests that refinements lead to small, but significant improvements to interaction speed [172].

7.5.3 Post-study Improvements

The results from our user study demonstrate the feasibility of our scoring model to optimize the layout of existing UIs. Based on impromptu feedback from three participants during the study and our observations on how users interacted with Reflow, we made some improvements to further minimize the amount of disruption to the original UI. Specifically, participants commented that the adaptation process affected the ordering of elements (*e.g.,* an element was moved past an adjacent neighbor) and caused controls in UI structures navigation bars became mis-aligned. We implemented additional heuristics to extract these constraints from the UI and used them further guide the optimization process.

Constraint Extraction: Our approach maintains a list of constraints represented as equations

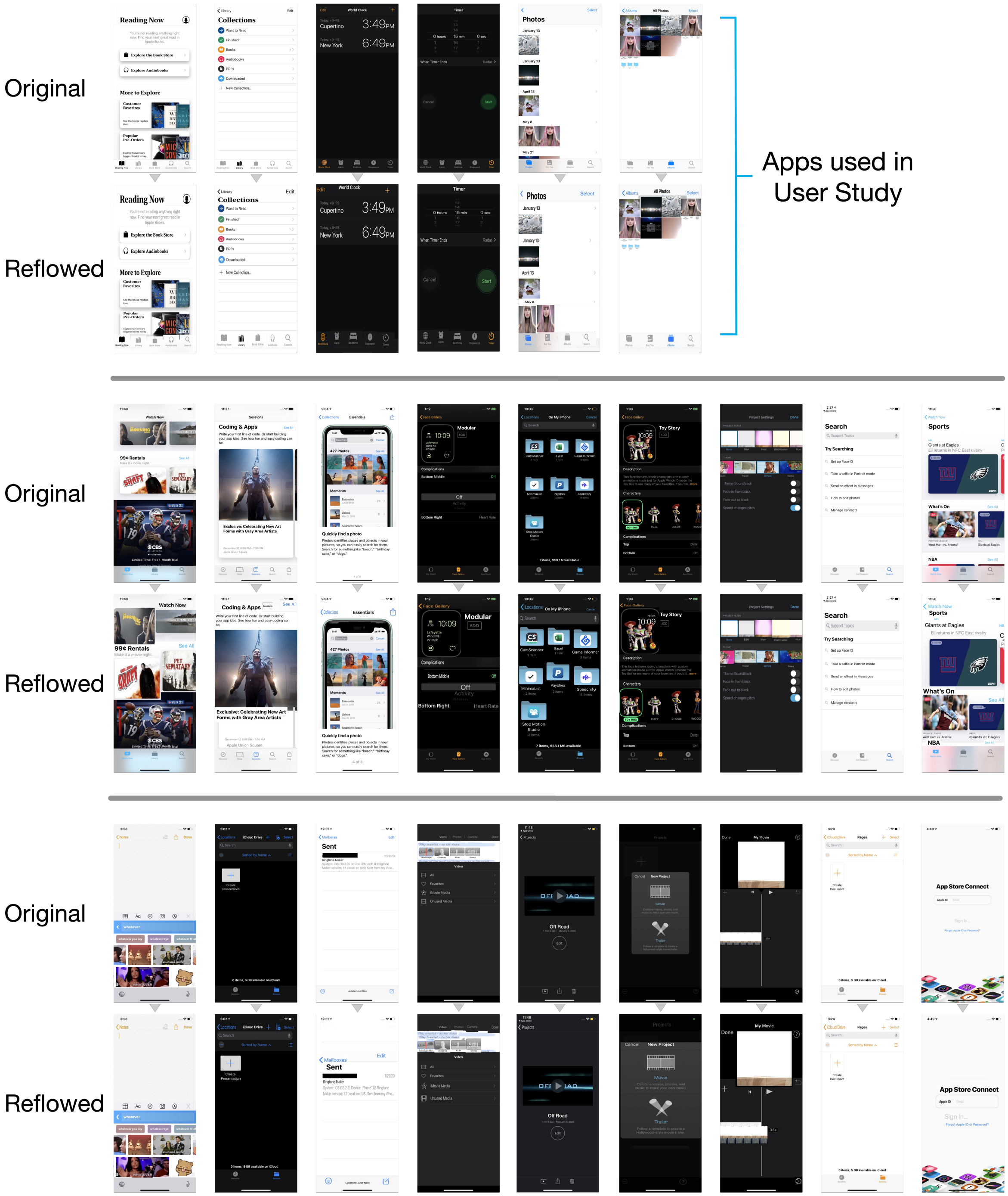


Figure 7.9: A gallery of app screens optimized by Reflow. For the apps used in the User Study, we only show the first and last screens of each app (each app has 5 screens). As was our intention, most changes by our system were conservative (resulting in little to no changes) and aimed at minimizing any negative effects introduced by drastic changes. Some text (*e.g.*, email address) is redacted in black.

and tests them against pairs of elements in the layout to determine if they hold. We check for two types of constraints: *(i)* relative positioning *(ii)* alignment.

The enforcement of relative positioning ensures, for example, that if an element A was to the left of another element B, it remains to its left. Relative positioning is represented using a multi constraint described by the following equation.

$$x_i + w_i \leq x_j \tag{7.5}$$

To reduce the number of relative constraints generated by our approach (making optimization more efficient), we remove redundant relationships through transitive reduction. For example, if A is left of B and B is left of C, the relative positioning of A and C is implied and can be omitted as an explicit constraint.

Alignment constraints require that some part of two elements are arranged on the same line. We consider 3 types of alignment between elements: *(i)* beginning alignment, *(ii)* center alignment, and *(iii)* ending alignment. These three types are computed for both the x and y dimension, resulting in a total of six possible element alignments. Note that most interface builders may also allow mixing of these types *e.g.,* the left edge of element A is aligned with the center of element B. For simplicity, we omit these.

Because our UI element detector introduces a small amount of error in the bounding boxes, exact computation of alignment (*e.g.,* computing $x_i = x_j$) would be inaccurate, leading to many undetected constraints. Thus, we allow alignments to be met inexactly using slack variables. We consider three types of alignment: beginning alignment, center alignment, and end alignment.

$$\begin{array}{ll} x_i + \epsilon_i = x_j + \epsilon_j & \epsilon_i \geq 0, \epsilon_j \geq 0 \\ x_i + w_i + \epsilon_i = x_j + w_j + \epsilon_j & \epsilon_i \geq 0, \epsilon_j \geq 0 \\ x_i + \frac{w_i}{2} + \epsilon_i = x_j + \frac{w_j}{2} + \epsilon_j & \epsilon_i \geq 0, \epsilon_j \geq 0 \end{array} \tag{7.6}$$

Constrained Optimization: We used the non-convex optimization technique introduced by Platt and Barr [256] to augment our original layout optimizer. Specifically, we added the heuristically extracted constraints as "secondary functions" of the main objective function (*i.e.,* minimizing expected difficulty) that the optimizer aims to minimize. Note that the while this technique allows the consideration of constraints, it does not guarantee that they will be met in the final solution.

7.6 Heuristic Evaluation

7.6.1 Procedure

To determine the acceptability of our improved system and to obtain qualitative feedback, we conducted a heuristic evaluation as described in Nielsen [242]. Heuristic evaluation has been previously been used as a design-centered analytic evaluation for user interfaces [36, 222]. In a heuristic evaluation, expert evaluators examine the interface or aspects of the interaction to identify usability problems. Nielsen [242] recommends the use of around three to five evaluators, so we contacted 10 people in our organization, in anticipation that some would be unavailable.

Table 7.2: Relative Likert Scores for Heuristic Evaluation

	AX			DS		
Heuristic	E1	E2	E3	E4	E5	E6
Alignment	−1		−1	−1	+1	−1
Selection Time	+1	+1	+1		+1	+1
Clutter		−1	−2	−1	+2	−1
Saliency	−1	+1		−1	+1	−1
Grouping	−1		−1		+1	

Expert evaluations from heuristic evaluation. Ratings are normalized to show deviation from the neutral option ("About the same (3)"). Positive scores indicate better change and negative scores indicate worse change. AX denotes accessibility expert while DS denotes design expert.

6 experts (3M, 3F) from different backgrounds (3 designers, 3 accessibility experts) agreed to participate and provided diverse feedback. Recruitment was done via convenience sampling—we reached out to potential evaluators using our organization's messaging software. All of our evaluators had multiple years of experience in their field and five had doctoral degrees in their respective areas of specialization.

The heuristic evaluation was conducted online, and evaluators were sent a link to the evaluation materials. The link first gave a brief description of Reflow and the evaluation task. Evaluators were instructed to watch a video clip that showed four usage scenarios (*e.g.,* while walking or holding a shopping bag with one hand) where a user interacted with app UIs. Original and adapted versions of the UI were overlayed on the video and allowed evaluators to assess different qualities of the UI in context. Evaluators could pause and replay the video as many times as they needed during the evaluation to more closely inspect aspects of the UI and usage scenario.

We provided a questionnaire with a set of UI heuristics for them to evaluate, informed by prior work on layout usability [284, 306]. We removed heuristics (*e.g.,* color harmony) that were not applicable to Reflow and settled on the following: alignment, selection time, visual clutter, saliency, and element grouping. For each heuristic, we provided a brief description and asked evaluators to rate adherence using a 5-point Likert scale for comparison ("Significantly Worse (1)" to "Significantly Better (5)"). We also asked evaluators to provide rationale for their ratings and encouraged elaboration on the positive and negative aspects on the layout changes introduced by Reflow.

7.6.2 Results

7.2 shows each evaluator's 5-point Likert-type item responses. Overall, the results of the heuristic evaluation confirm that Reflow improves touch efficiency while minimally disrupting the user experience. We summarize the feedback from our evaluators along each of the heuristics.

Alignment: *Heuristic.* Alignment refers to the internal alignment of elements with each other. *Evaluation.* Most evaluators agreed that Reflow's refinements slightly impacted alignment. While our improved layout optimizer does detect and account for UI constraints such as alignment, it does not guarantee that these constraints are met. We found that some evaluators' intepretations of "alignment" included aspects such as spacing (not explicitly handled by our system) as well as edge alignment (handled by our system) ($\mathrm{E3_{AX}}$). Nevertheless, $\mathrm{E2_{AX}}$ observed that some layouts *"probably could change more without hurting understanding,"* indicating that Reflow's refinements kept semantic relations are kept intact.

Selection Time: *Heuristic.* Selection time refers to the time needed to select the target element. Since the evaluators did not directly operate each UI, we asked evaluators to estimate the selection time from the relative positioning of elements (*i.e.,*), as in previous work [284, 306]. *Evaluation.* Feedback from our evaluators mostly indicated that our adaptations allow for more rapid selection times during use, but evaluators expressed some uncertainty as they were estimating purely from visual appearance ($\mathrm{E1_{AX}}$, $\mathrm{E3_{AX}}$). We did indeed empirically measure selection times from our user study (7.1), and our expert's estimates are consistent with our findings. $\mathrm{E5_{DS}}$ pointed out that our system allowed faster interaction *"without having to use reachability"* and would *"certainly save time,"* since performing the Reachability gesture (7.3) would require more time than using Reflow.

Taken together with other feedback (comment by $\mathrm{E2_{AX}}$ on alignment), this suggests that supporting manual adjustment of optimization levels may help Reflow better serve users with differing goals—either prioritizing selection time or preserving aspects of the original UI (*e.g.,* alignment).

Visual Clutter: *Heuristic.* Visual clutter refers to how confusing a display is. The more cluttered a display, the more difficult it is for an element to catch a user's attention *Evaluation.* $\mathrm{E2_{AX}}$ pointed out that movement of semantically-important elements contributed to confusion. There was some level of disagreement between evaluators ($\mathrm{E3_{AX}}$, $\mathrm{E5_{DS}}$), which was in part influenced by different interpretations of the heuristic. For example, $\mathrm{E3_{AX}}$ attributed the higher levels of visual clutter to *"worse alignment, unexpected/unintuitive negative space, and overlapping elements,"* indicating the consideration of multiple factors such as alignment, spacing, and overlap.

Saliency: *Heuristic.* Saliency refers to the degree to which important elements are more likely to catch the attention of the user. *Evaluation.* Evaluators suggested that our system's modifications did not induce significant changes in saliency, which is desirable since a large saliency change would imply a violation of design intent. $\mathrm{E3_{AX}}$ indicated that the refinement process only resulted in *"subtle changes"* and did not make a single element any more or less salient than the others. $\mathrm{E2_{AX}}$ noted that size refinements, which allows elements in "difficult areas" to be selected more easily, positively impacted the saliency of small items by enlarging their tap targets; however the same behavior was viewed negatively by $\mathrm{E1_{AX}}$: *"the adaptations didn't have much impact on saliency, so I almost selected about the same, but I noticed that the tabs have increased size so they may be over-emphasized."*

Element Grouping: *Heuristic.* Elements that are clustered near each other are perceived as a unified object. *Evaluation.* Element grouping was also minimally affected by Reflow's refinement process. While evaluators generally agreed that element grouping remained "about the same" ($E2_{AX}$, $E4_{DS}$, $E6_{DS}$), some pointed out instances where the adjustment of element spacing led to ambiguity: *"on the 4th screenshot pair, the inbox and arrows have scrunched down too closely to the App Store and To field, leaving not enough separation"* ($E1_{AX}$).

7.7 Discussion

7.7.1 Flexible Personalization through Difficulty Maps

An architectural decision choice we made in Reflow was to parameterize the difficultly map as an explicit input to the neural scoring model (7.3). Doing so has a number of advantages.

Difficulty maps are the key mechanism through which we enable personalization, because they allow users to recalibrate their touch interactions without having to retrain the neural scoring model for every user. Having to retrain a model for every user is not practical, especially if the retraining must be done on a mobile device.

An second advantage of parameterizing difficulty maps is that Reflow can be easily extended to other types of interactions. While our studies evaluated one-handed touch interactions, the Reflow is not restricted to this. Difficulty maps can also be constructed to support reachability, handedness, motor impairments, or other touch accommodations needed to support users. Difficulty maps can even go beyond touch interactions, for example, when using a pen input or other pointing device. Once a mode of interaction is translated to a difficulty map, the rest of the Reflow architecture can take advantage of this—without requiring any changes the remaining stages in the Reflow pipeline.

Finally, difficulty maps make it possible to offer users a set of pre-defined profiles for common touch interaction scenarios. Pre-defined maps could cover scenarios such as one-handed usage and allow many users to benefit from adaptation without having to perform manual calibration. Should these pre-defined maps not support the user, they can always fallback to performing a quick calibration to construct a personalized difficulty map.

7.7.2 Design Space between Touch Efficiency and Layout Preservation

Reflow applies *refinements*—small UI adaptations that minimally disrupt the UI layout—as a design choice for performing layout optimization. Our heuristic evaluation validates that this decision choice as an appropriate one. However, this decision choice is only one possible point across the full design space, which includes a spectrum of trade-offs between improving touch efficiency against preserving the existing layout. Users may have different preferences along this design space.

For example, all evaluators in our heuristic evaluation—except E4—indicated that selection time would be improved by Reflow. E4 desired to see more dramatic improvements to touch interactions, even if this would require Reflow to make more substantial disruptions to the layout. In contrast, E3 disliked the clustering that resulted from Reflow's layout optimization. Users such as E3 may find it acceptable to have fewer touch improvements, if this would result in smaller layout disruptions.

One possibility is to give users more control over this design space. Consider a slider control that allows the user to select points in the design space between maximizing touch interaction efficiency, minimizing layout disruption, or some balance in between these two extremes. Users may also want to selectively enable or disable specific types of optimizations, for example, they may prefer not to allow the size of UI elements to change. Users may even have different preferences between applications, with varying expectations about layout disruption for the different applications.

7.7.3 Opportunities for Reflow

Through our prototype implementation and experiments, we identified opportunities and future work for Reflow, including *(i)* extracting and incorporating screen semantics, *(ii)* improving interaction fidelity, and *(iii)* studying the effect of applying refinements over time.

Extracting Screen Semantics

Several of these opportunities involve addressing existing limitations of Reflow. First, the set of constraints that we automatically extract should be extended to support those available in conventional UI authoring tools, such as vertical and horizontal alignment guides, distributing vertical and horizontal spacing, and resizing elements across axes. Including these constraints and inference techniques [44, 157, 168, 217] in Reflow would further minimize layout disruptions. Currently, Reflow internally represents UIs as a list of bounding boxes, but the limitations of this approach are that it is unable to capture semantic relationships *between* elements during inference. Because of this restriction, Reflow currently treats a list of n UI elements as n unrelated UI elements, which is potentially a lost opportunity for layout optimization.

Improved Interaction Fidelity

Reflow's pixel-based pipeline consumes an image (*i.e.,* screenshot) of the current UI as input and also generates an image as output. The refined UI is made interactive by making certain parts of the output image respond to touch events, which can then be forwarded to the original app. While this allows Reflow to be applied to any screen, it may lead to poor performance on screens with dynamic properties such as animated content and scrolling because this behavior cannot be adequately captured in a static screenshot.

One way to extend the current approach to handle these cases is to perform this image-to-image process multiple times per second, thus updating the output as frequently as dynamic behavior occurs. Besides the optimization and performance challenges this entails, repeatedly optimizing single video frames as independent inputs may lead to artifacts such as jitter. An alternative approach is to regenerate the interface from extracted semantics and interfaces, which has previously been applied to web applications [240]. Recent work in pixel-based semantic extraction [104, 326] suggests this may also be possible for mobile UIs.

Extended Evaluation

Finally, a longer-term usage study may reveal more detailed effects of adaptive UIs and, more specifically, our refinements approach. Although our studies demonstrate the effectiveness of refinements for improving touch efficiency, it would be important to evaluate Reflow in longitudinal studies. The benefits of Reflow can only be fully realized through cumulative use: the longer users use Reflow, the more opportunities they have to take advantage of the adapted UIs. As Gray and Boehm-Davis [124] observe—"milliseconds matter"—and even seemingly small improvements add up over frequent and repeated touch interactions.

7.8 Conclusion

In this paper, we introduced Reflow, a system that automatically applies small, personalized UI adaptations—called *refinements*—to mobile app screens to improve touch efficiency. Reflow supports real-world UIs without any source code or metadata dependencies through pixed-based element detection. Reflow optimizes a UI by *(i)* extracting its layout from its screenshot, *(ii)* refining its layout, and *(iii)* re-rendering the UI to reflect these modifications. We conducted a user study with 10 participants and found that UIs optimized by Reflow were on average 9% faster to use. We conducted a heuristic evaluation with 6 experts to elicit qualitative feedback about Reflow and validate that the system's UI refinements improve selection time while minimizing layout disruption. The results of our work demonstrate that refinements applied by Reflow are a useful UI adaptation technique to improve touch interactions.

8 UICoder: Finetuning Large Language Models to Generate User Interface Code through Automated Feedback

8.1 Preamble

Content in this chapter is based on a conference paper that has been accepted to NAACL 2024 [332].

In Chapter 7, I presented a system that can modify any existing app at runtime using only its pixels as input. Reflow applied small changes, called refinements, to an existing app's layout and re-rendered the resulting interface for end-user use. Our experimental results show that for a limited set of apps and users, even small changes could lead to performance improvements, *i.e.,* an average of 9% faster selection time. However, Reflow was limited in several ways. First, it used a primitive approach to re-generating app interface by cropping and re-arranging parts of the original screenshot. This prevented the system from applying substantial changes, since it could increase the likelihood of rendering artifacts. Secondly, while the output was interactive, it was also a primitive example of an interaction proxy [345], as it consisted of a re-rendered image with touch areas re-mapped to the original UI. This made the system inefficient for apps that contained interactive elements, since the output would need to be continually refreshed to stay in-sync with the state of the original interface. Based on these observations, I focused on developing systems that can support the dynamic generation of higher-quality UIs through compilable code. I first investigated this approach in the context of description-based UI code generation, since it is by itself a challenging and unsolved problem.

LLMs struggle to consistently generate UI code that compiles and produces visually relevant designs. Existing approaches to improve generation rely either on expensive human feedback or distilling a proprietary model. In this paper, we explore the use of automated feedback (compilers and multi-modal models) to guide LLMs to generate high-quality UI code. Our method starts with an existing LLM and iteratively produces improved models by self-generating a large synthetic dataset using an original model, applying automated tools to aggressively filter, score, and de-duplicate the data into a refined higher quality dataset, and producing a new LLM by finetuning the original on the refined dataset. We applied our approach to several open-source LLMs and compared the resulting performance to baseline models with both automated metrics and human preferences. Our results show the resulting models outperform all other downloadable baselines and approach the performance of larger proprietary models.

8.2 Introduction

Designing and implementing user interfaces (UIs) has traditionally been a difficult and time-consuming process that requires expertise and effort. Because of this, there are significant barriers for designers to quickly prototype design candidates, developers to implement working apps, and end-users to create customized interfaces. Large language models (LLMs) present a promising solution since they are trained on large amounts of natural language text and code, which allows them to relate high-level user specifications to concrete code implementations. Following a large-scale unsupervised "pretraining" phase, the model is finetuned to perform specific tasks, such as generating code, based on natural language directives in a process called instruction-tuning. Through this process, LLMs have been trained to be proficient in conversation, develop reasoning abilities, and even use external tools [52, 275].

Nevertheless, it is still difficult for LLMs to reliably generate syntactically-correct, compilable code for UIs. Most of the examples found in crawled web pages and repositories are not self-contained or are of low quality [128]. Even in curated or manually authored finetuning datasets, examples of UI code are extremely rare, in some cases making up less than one percent of the overall examples in code datasets [233]. Furthermore, by their nature LLMs are unable to directly incorporate visual or spatial feedback into their training process, which would seem to be an important aspect of UI design and implementation.

In this paper, we describe an automated method for training LLMs to generate UI code from textual descriptions. We specifically focus on training models to implement UIs using the SwiftUI framework, though our method would likely generalize to other languages and UI toolkits. Instead of relying on additional external data, our approach finetunes LLMs to generate improved UI code entirely from their own previous outputs. We first prompt an existing LLM to generate a large synthetic dataset of SwiftUI programs from a list of UI descriptions. We then use a compiler and vision-language model [260] to aggressively score, filter, and deduplicate the output samples to create a refined higher quality dataset. By finetuning on the subset of high-scoring outputs, an improved LLM learns to generate UI code that *i)* successfully compiles and *ii)* is relevant to the input description. During subsequent iterations, the improved LLM generates higher-quality datasets, which results in further performance gains.

We call our resulting model UICoder, because we originally started with the open source LLM StarCoder [190]. In this paper, we use the latest instruction-tuned version of StarCoder, at the time of writing, called StarChat-Beta, as our base model [311]. We applied five iterations of our method, resulting in nearly one million generated SwiftUI programs, and used preference-based alignment methods to train three models for text-to-UI code generation. In a series of experiments that measure both automated and human preference, we show that our models significantly outperform other downloadable baselines and approach the performance of much larger proprietary LLMs. We think our results are particularly impressive because our models originate from StarCoder, and Swift code repositories were accidentally omitted from the training of this model [190]. The finetuning dataset used to create StarChat-Beta from StarCoder contains just one Swift example out of ten thousand total examples.

To summarize, the contributions of our work are as follows:

1. We introduce an automated method for generating description-to-code datasets for UIs by using code compilers and vision-language models to score and filter self-generated data.
2. We applied five iterations of our method to train the UICoder model for generating complete SwiftUI implementations from natural language descriptions. Our experiments show that UICoder *i)* outperforms all other downloadable models and *ii)* approaches the performance of larger proprietary models.
3. We show that incorporating the synthetic dataset generated by UICoder significantly improves other LLMs' UI code generation capabilities, without needing to undergo multiple iterations of the training process themselves.

8.3 Training Procedure

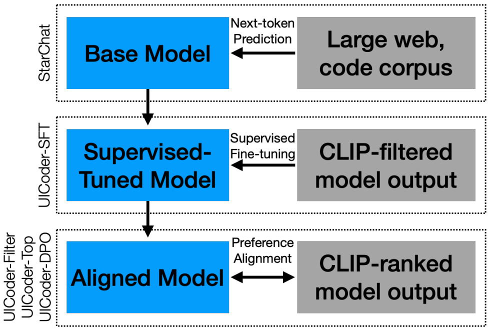


Figure 8.1: A flow chart showing an overview of the multi-step training process of our model. Our process is based on prior LLM finetuning approaches [250] and consists of a base model, supervised-tuned model, and an aligned model. Different sources of data and training techniques are used at each stage.

In this section, we detail the training procedure used to train LLMs to generate SwiftUI code for a UI given its natural language description. SwiftUI is a toolkit for the Swift language that allows cross-platform UIs (desktop, tablet, mobile, and watch) to be composed through a domain specific language (DSL). Generating SwiftUI using LLMs is difficult, due to challenges associated with code generation in general [65], and especially poor representation of SwiftUI programs in publicly available training data and evaluation benchmarks.

To improve the generation capabilities of LLMs, we apply a training procedure based off of previous work [250] that involves three high-level stages. Our approach is unique in that it uses automated feedback from code compilers and multi-modal visual-language models in place of human annotations. Figure 8.7 shows the overall overview of our training approach that involves *i)* training (or using a pre-existing) base model, *ii)* using supervised finetuning, then *iii)* preference alignment techniques to further improve performance.

8.3.1 Training Datasets

To train our model, we used several UI datasets: *i)* Screen2Words [315], *ii)* AMP [347], and *iii)* Crawls [100]. Between them, there are a total of 800,000 iOS and Android UIs. In this work, we primarily focused on generating code for mobile UIs, although additional datasets (e.g., web [170, 330]) could be incorporated in the future. The datasets we used mainly contained screenshots, which were sometimes paired with natural language descriptions or other metadata. In our training procedure, we only focus on the screenshot images and natural language descriptions. Note that these datasets do not contain the source code of the UIs; therefore, the model must learn to match its own generated code output to relevant examples. We describe two additional measures we took to increase the number and complexity of descriptions used for training.

Generated Descriptions. The AMP and Crawls datasets did not contain natural language descriptions of UI screenshots, so we used a large visual-language model (VLM) [189] to generate descriptions of screenshots. While previous work has generally found off-the-shelf image captioning models to be ineffective at captioning UI screenshots [183], we found that a combination of stronger models, prompt engineering and strict filtering were sufficient for weak labeling of these screens.

LLM-assisted Augmentation. The human-annotated descriptions in the Screen2Words dataset were often very simple (*e.g.,* incomplete sentences) and underspecified, and we observed that this led to simple outputs as well. To add more detail, we used an additional open-sourced LLM (the Falcon-7B model [26] finetuned on open instruction-tuning data [165]) to paraphrase and add more detail to the original description text. In total, we used two prompts to generate 200,000 alternate descriptions for the Screen2Words dataset. Since the LLM used in this process did not have access to the original screenshot, it is possible that it can “hallucinate” inaccurate details. However, we found that our filtering methods could minimize inaccurate descriptions since samples are excluded if there isn’t a strong match with the original screenshot.

8.3.2 Base Model

The first step of our approach is to train or use a pre-existing base model (see Figure 8.7). Training base models is a resource-intensive and time-consuming process that involves unsupervised training of an LLM on a very large corpus, often containing billions or trillions of tokens. Therefore, we chose to use an existing pre-trained 15B parameter model, StarChat-Beta [311], as a starting point, which was the best available open model at the time we started the project. StarChat-Beta is an LLM based on StarCoder [190] which was trained primarily on *i)* TheStack [190], a large dataset (250B tokens) of permissively licensed code repositories, *ii)* crawled

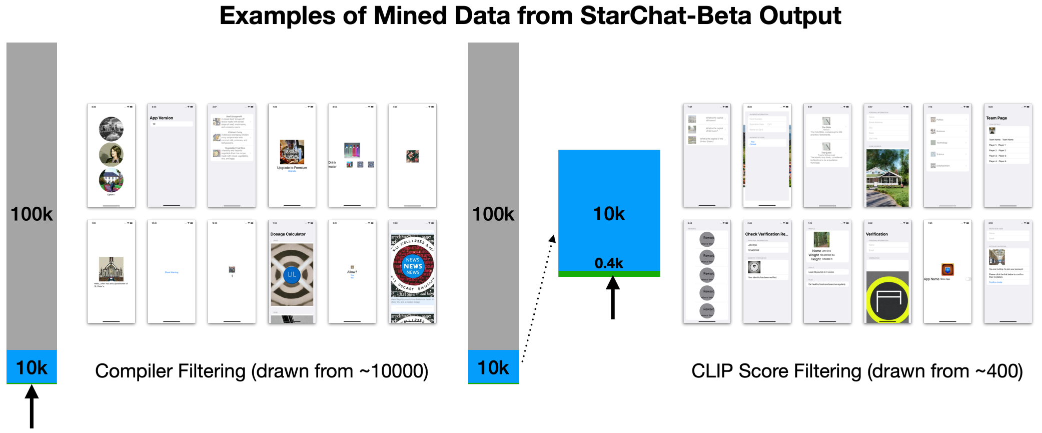

Examples of Mined Data from StarChat-Beta Output
100k
10k
Compiler Filtering (drawn from ~10000)
100k
10k
10k
0.4k
CLIP Score Filtering (drawn from ~400)

Figure 8.2: A collection of randomly drawn samples from 100,000 SwiftUI programs generated by the StarChat-Beta base model. Only around 10% of the generated samples are compilable, and we rendered them into screenshots using Stable Diffusion to generate image assets. On the left, we randomly sampled twelve screenshots that passed the compilation filter and show that many of the screens are very simple. On the right, we randomly sampled twelve screens that passed the CLIP score filter, which are more complex and of higher quality.

web pages, and *iii)* OpenAssistant-Guanaco [83], a small instruction-tuning dataset. Notably, StarChat-Beta's training data contains little to no SwiftUI data [190]. Swift code repositories were excluded by accident when creating TheStack dataset [190], and upon manual inspection, we found that the OpenAssistant-Guanaco dataset only contains one example (out of ten thousand) with any Swift code in the response field. We hypothesize that any Swift examples seen by StarChat-Beta during training were most likely from crawled web pages, which are possibly lower quality and less structured than repository code.

8.3.3 Supervised Finetuning

Since most "base models" are trained through next-token-prediction (i.e., text completion), they are unable to respond directly to certain types of instructions. During the supervised finetuning stage, a base model is adapted to perform tasks or respond in a certain format by training on examples of input/output pairs.

An overview of our approach is shown in Figure 8.3, where we generate the input/output pairs needed for supervised finetuning through automated feedback. Our approach is similar to "rejection sampling" techniques, where high quality examples are mined through repeated sampling (i.e., generating output candidates), scoring, and selection (i.e., rejecting samples that do not meet a certain criterion). Figure 8.2 illustrates parts of this process applied to StarChat-Beta and provides some intuition for why this approach works. While the StarChat-Beta base model has poor performance overall for SwiftUI code generation, frequently producing uncompilable code or very simple UIs, we successfully detected a subset of higher-quality examples using compilation success and CLIP score as filtering conditions. The initial proportion of "mined" examples is extremely small (400 out of 100,000). However, the model's performance improves

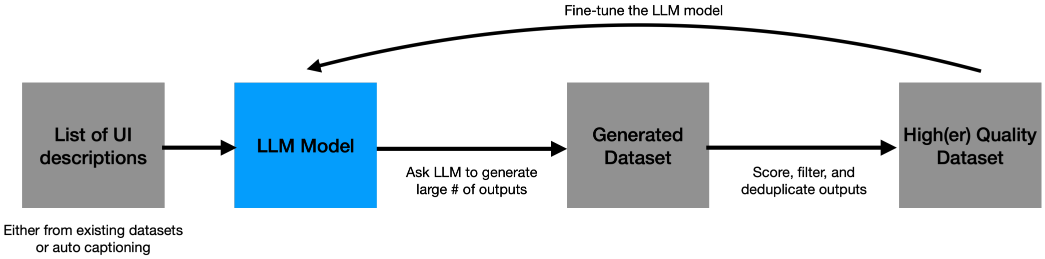

Fine-tune the LLM model
List of UI descriptions
LLM Model
Generated Dataset
High(er) Quality Dataset
Ask LLM to generate large # of outputs
Score, filter, and deduplicate outputs
Either from existing datasets or auto captioning

Figure 8.3: A flow diagram representation of the filter-then-train approach used for supervised finetuning. A list of descriptions is fed into an LLM model, which is used to generate a synthetic dataset. The generated dataset is scored, filtered, and de-duplicated to improve its quality. This data is used to finetune the LLM model, which restarts the process.

with each iteration of training on mined examples. As a result, it progressively generates a higher percentage of high-quality data that pass the filter. We repeated this process four times starting with the StarChat-Beta base model. We name the finetuned model *UICoder-SFT*, to reflect that it is a “descendant” of the StarCoder [190] model, and it has undergone supervised finetuning (SFT).

Sample Generation.

We used a randomized strategy for sampling and generating code, where the program selected random screen descriptions from our training datasets fed them into the model to generate code. Selection was done independently, resulting in some prompts having multiple outputs and others being skipped altogether. During each training iteration, we allocated roughly a week for sample generation, which resulted in around 200,000 total samples.

Data Filtering.

We used three methods to filter generated data for high-quality finetuning examples: *i)* compilation success, *ii)* CLIP-based output relevance, and *iii)* de-duplication. Early iterations had many examples filtered out due to compilation failure since the the base model was not effective at generating syntactically-correct SwiftUI code. During later iterations, more samples were rejected due to output relevance and the de-duplication filter. The total number of mined examples also increased over time, due to more of the generated samples fulfilling the requirements. We also adjusted the hyperparameters of some filters to be more selective over time, to obtain higher quality training samples and also due to memory constraints.

Compilation Success. We keep only fully compilable programs (compile warnings are ignored). This is necessary because filters used in later stages rely on the rendered output of the SwiftUI code, which is not possible to generate without a working program. However, compilation success alone is not a sufficient metric for output quality and introduces biases into the dataset. In our early experiments, we attempted to finetune our model without any other filtering steps and identified failure cases: *i)* the model produced only simple outputs, since programs with fewer lines of code are less likely to contain errors and *ii)* the generated outputs were not relevant to the input description. These observations led us to the development of additional features to counteract these biases.

CLIP Score. The CLIP score filter uses a vision-language model [151] to assign a numerical score to each input/output pair that measures how well the generated UI matches the input description. We constructed a natural-language prompt template that included tags such as "screenshot of a mobile app. award winning design." and the input description itself. The CLIP model was used to encode both the constructed prompt and screenshot into an embedding space, where a score was computed using the cosine similarity of the embeddings. Finally, we used a percentile-based threshold to keep the samples with the highest scores.

Based on early observations, we found that CLIP's matching score had drawbacks. One common error that we noticed was attributed to the model's tendency to give a high matching score if the image contains the original prompt text, due to the model's ability to "read" or recognize text robustly inside of images. These errors, sometimes called "typographic errors" [120], resulted in failure cases where the model included a Text element with words from the original prompt. To address this problem, we modified the CLIP score to also incorporate the original ground-truth screenshot as a part of the calculation. While the goal is not to reconstruct the ground-truth screenshot, it helps avoid typographic errors by encouraging the model to produce screenshots similar to realistic samples.

De-duplication. Filtering by compilation success and output relevance alone may still result in biases in the generated data. We performed data de-duplication to remove examples that resulted in highly similar outputs. We used a density-based clustering algorithm [94], which is a type of clustering algorithm that doesn't require prior knowledge of # of clusters, to group examples based on the CLIP embeddings of their rendered screenshots. This process makes it easier to identify groups of outputs that result in a highly similar appearance or layout. For each identified cluster, we keep only the example that has the highest CLIP score.

8.3.4 Preference Alignment

Following supervised finetuning, LLMs often undergo an additional *alignment* stage to better match with human preferences or meet specific criteria like helpfulness and harmlessness. Unlike supervised finetuning, where the model is trained to match pre-determined outputs for each input, alignment techniques allow the model to generate its own output candidates, and then provide a numeric reward or pairwise preference labels as training signals. We hypothesized that our model could benefit from this process by implicitly learning to "prefer" output candidates with high reward values while avoiding those with low CLIP scores or compilation errors. Unlike previous work [250, 308] that used human ratings for alignment, we explore the use of rankings generated using the Swift compiler and the CLIP model.

Ranked Sample Generation.

To generate data for preference modeling, we modify our data generation strategy to produce paired examples, which in our case is an input prompt paired with several possible code implementations. It was difficult to apply this generation strategy from the the beginning of the supervised finetuning stage since it requires that the model has a sufficient compilation rate to generate multiple compilable outputs for the same input prompt. After randomly selecting a screen description, we use the LLM to generate 10 outputs for the same input through a set of manually created sampling configurations. Note that while it is possible to repeatedly sample the model with the same configuration, we found that more output diversity could be achieved with different sampling profiles.

Once multiple outputs are generated for each input, they are ranked using pairwise rules, where higher rankings indicate higher quality.

- All compilable samples are ranked above non-compilable samples.
- Compilable samples are ranked by their CLIP score.
- Non-compilable samples are ranked by the number of error-free lines (i.e., lines that do not contain a compiler error) divided by the total number of lines in the program.

Modeling Approaches

We used three different modeling approaches to further finetune UICoder-SFT with the generated preference pairs, which resulted in three variations of the model: UICoder-DPO, UICoder-Top, and UICoder-Filtered.

UICoder-DPO was trained using an algorithm called Direct Preference Optimization (DPO). At a high level, DPO involves repeatedly selecting a pair of outputs for each training input - a “chosen” output that is ranked higher than a “rejected” output.[261]. The DPO algorithm repeatedly selects these pairs, and the model is trained to estimate a reward value that is higher for the preferred output. The DPO algorithm requires significantly more GPU VRAM than supervised finetuning, so we used 4-bit quantization with the QLoRA technique so that it could fit on a single A100 GPU [83].

UICoder-Top was trained using the previous stage’s objective, but with the top output for each input as the target. Unlike the previous filtering method which skipped “hard” input examples, this approach exposes the model to all input prompts during training, which we hypothesized could be more effective.

Finally, as a point of comparison, we applied one more iteration of the previously used filter-then-train algorithm that ignored rankings to the newly generated data. We refer to this model as UICoder-Filter.

8.4 Training Infrastructure

In this section we provide a description of the infrastructure built to support model training. Our setup *i)* generates large volumes of SwiftUI code, *ii)* renders the SwiftUI code to screenshots, and *iii)* scores, filters, then de-deduplicates the synthetic dataset before finetuning our model. The servers are connected to a cloud storage provider that allows them to share files between each other.

Code Generation Server. The code generator is used to generate a large number of SwiftUI programs using our model. The code generator orchestrated a fleet of worker nodes each of which each downloads the latest model weights from cloud storage. Since we used a pool of servers shared among our organization, the number of worker nodes varied throughout the

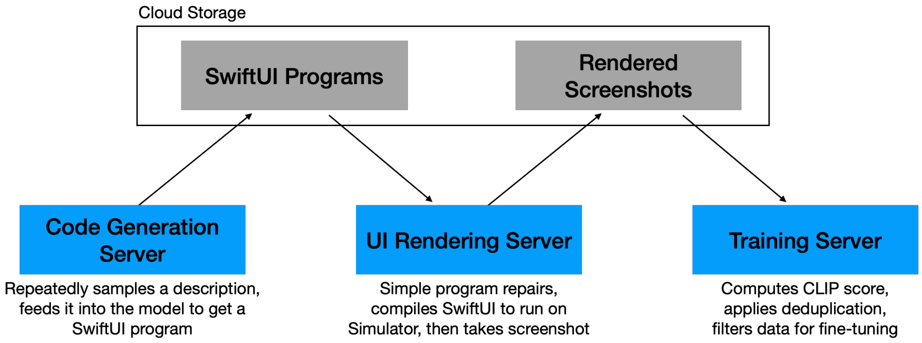

Cloud Storage
SwiftUI Programs
Rendered Screenshots
Code Generation Server
Repeatedly samples a description, feeds it into the model to get a SwiftUI program
UI Rendering Server
Simple program repairs, compiles SwiftUI to run on Simulator, then takes screenshot
Training Server
Computes CLIP score, applies deduplication, filters data for fine-tuning

Figure 8.4: Training infrastructure flow diagram that shows the role of the code generation server, UI rendering server, and training server.

training process. Generally, we had roughly 40-50 workers running continuously, each equipped with multiple NVIDIA P100, V100, or A100 GPUs. [1]

Each code generation node repeatedly sampled a description of a UI from our training datasets using uniform random sampling and constructed a prompt using our model's prompt template. A SwiftUI program was then generated by sampling our model until a pre-defined stop token was reached. Under some training configurations, multiple outputs were generated from the same description. Because programs were generated using stochastic, temperature-based sampling, this could potentially result in different variations of a UI. Generated SwiftUI programs were periodically uploaded to network storage.

UI Rendering Server. The UI renderer was used to convert SwiftUI programs to rendered screenshots. It consisted of a main routing server that coordinated multiple worker nodes and image asset renderers. Each worker node runs macOS with XCode and the iOS simulator that repeatedly *i)* downloads queued SwiftUI programs, *ii)* renders them to screenshots, then *iii)* uploads the results to the network storage. The image asset renderers ran text-to-image models that generated image assets referenced by the SwiftUI code. The SwiftUI program is first run once to determine the required image assets by parsing the error logs for missing image asset names. The asset names are used to generate image assets, which are included in the project for a subsequent run.

Before program compilation, we applied automated program repairs (APRs) to try to automatically fix common errors. While LLMs can be used to find and resolve bugs [98], we found that current open-source LLMs *i)* could not effectively repair bugs associated with compiler errors and sometimes even introduced additional errors into the code, *ii)* made large changes to the original code including sometimes completely rewriting it, and *iii)* were slow since they required running another inference through a large model. We focused on lightweight, heuristic repairs based on regular expressions, which we hypothesize to be useful for many types of simple code transformations. We manually authored a small number of repairs that are not meant to fix *all* errors in a given program, but are able to identify and repair certain types of bugs with high precision. We used a combination of *matching-based repairs* and *compiler-guided repairs*.

[1] Worker nodes with P100 or V100 GPUs ran the inference with the `float32` data type to support the full precision required for the dynamic range of the model's native `bfloat16` floating point format.

Matching-based repairs used regular expressions to match common errors in the source code, such as whitespace errors caused by tokenization, and correct them. Compiler-guided repairs used the compiler errors output by the Swift compiler to localize and fix errors through manually written heuristics. This process was slower because it involved running the compiler multiple times.

Training Server. The model trainer downloaded and collated the *i)* input descriptions, *ii)* generated SwiftUI programs, and *iii)* the rendered UI screenshots. A text-image matching model [151] was used to embed score each rendered UI screenshot by computing the cosine similarities between *i)* the embedded input description, and *ii)* the embedded reference screenshot.

Depending on the training stage, different algorithms for dataset generation and model training were applied. During the earlier training phase, a subset of high-scoring examples including a description paired with a code implementation were used supervised finetuning. During the later phase, the scores were used to generate a dataset of ranked outputs, now a description paired with two ranked code implementations, for preference-based training. All models were trained using LoRA method [143] to improve training speed and efficiency.

When the model training finished, the updated model was uploaded to the network-backed storage so that the code generator used the newest version of the model for the next iteration of the training loop.

8.5 Experiments

We conducted experiments to *i)* measure the performance of our model over time, *ii)* measure the impact of our data on different LLMs and *iii)* compare against the performance of other baselines.

8.5.1 Evaluation Dataset

To support our experiments, we created an evaluation dataset of 200 UI descriptions, which is roughly the size of other LLM coding benchmarks [65]. We created an evaluation set by randomly selecting 100 screenshots from a held-out set of Android screenshots and 100 screenshots from a held-out set of iOS screenshots. We chose to construct the evaluation set by annotating randomly sampled screenshots instead of asking people to freely write their own list of descriptions. We did this to increase the diversity of tested prompts and the match the distribution of screenshots that are found in everyday apps, rather than focusing on the easiest ones to recall. We acknowledge that a drawback of this approach is that may not match the frequency of prompts actually fed into our system during real-world use, and we leave the longitudinal evaluation that would be needed to accurately collect these prompts to future work.

Each screenshot was manually annotated with a description that usually consisted of 1-3 sentences (mean 23.6 words, min 8 words, max 71 words). Compared to screen descriptions in existing datasets [315], our descriptions were longer and contained more variation since our guidelines were less rigid (i.e., no constraints on sentence structure or description length). There weren't any exact string matches between the descriptions in the evaluation set and the descriptions from the training set, but there may still be n-gram overlap (e.g., "sign-in screen of a fitness app" and "sign-in screen of a banking app") or other semantic similarities (e.g., "login screen" and "sign-in screen"). We consider this acceptable, since many screens are variations of each other and are built to achieve the same goal, and eliminating all similar descriptions would prevent the model from seeing a large class of examples during training (e.g., eliminating all sign-in screens from the training data).

For each tested model or API, we generated and rendered one program for every input in our evaluation set using the default sampling parameters and prompt template of that model, which we collected from their official web demos. If no official web demo could be located, we used the UICoder sampling parameters, which led to reasonable performance. Since the purpose of the evaluation is to evaluate the generated code and layout, we replaced all image assets with the same placeholder image: `Image(systemName: "photo")`.

8.5.2 Metrics

Measuring the quality of generated UIs is challenging, and to the best of our knowledge there is no automated benchmark that can evaluate description-based UI code generation, unlike for general-purpose coding where benchmarks such as HumanEval [65] may be used. One source of difficulty for UIs is that there is no single expected source of output that is used in common unit-testing approaches for code generation. Previous work has used objective functions grounded in cognitive principles [246, 247, 306] to measure UI layout quality, however they do not measure performance related to code or adherence to the input prompt.

In our work, we used a combination of metrics to evaluate different aspects of UI code: *i)* compilation rate, *ii)* CLIP score, and *iii)* a numeric Elo score based on human preference ratings.

Compilation Success. Compilation success was measured by calculating the ratio of compilable programs from the 200 input descriptions found in the evaluation set. Programs were compiled using the iOS version and Swift compiler included with Xcode 14. A high compilation rate suggests that the model has a good mastery of the syntax needed for generating correct code.

CLIP Score. Similar to our training procedure, we use the CLIP score as a fast, automated method to evaluate output relevance and quality. Previous work has shown a correlation with human judgments when evaluating text-to-image generation [136], and the CLIP score is more reproducible since it doesn't rely on human ratings, which may vary between people or over time. For evaluation, we used a larger and more accurate CLIP model than the one used in our training pipeline (OpenCLIP ViT-G/14 vs ViT-B/32 [151]), since the memory and efficiency constraints that applied during training were not relevant during evaluation. A high CLIP score suggests that the model's outputs are relevant to the input description.

Human Preference Elo. Following recent LLM evaluation techniques [348], we used pairwise human ratings to calculate Elo ratings for evaluated models [92]. This method is preferred over asking annotators for absolute ratings or full rankings because it greatly reduces the cognitive load but requires more samples for a full comparison. Each model starts out with an initial Elo rating, which is set to 1000, as done in prior work [348]. Pairs of models were randomly chosen and their rendered outputs in the form of screenshots were presented to six human raters who selected a preferred output or declared a tie. Raters were unable to see the name of the models that generated the outputs. If one of the models' code did not compile, it was automatically marked as a loss, and if both did not compile, a tie was recorded.

In total, around 3000 pairwise comparisons were recorded. For each comparison, the score of the “winner” was increased while the score of the “loser” was decreased based on the prior rating gap between the two compared models. This method resulted in a calibrated rating score where the difference of two models’ Elo ratings can be used to predict their “winning” probability vs. any other model. The Elo rating encompassed both aspects of code correctness and output quality and could be used to compare two models’ overall performance. For models with low compilation rate, their Elo score was primarily determined through the compilation check; however, it is unlikely that they could generate complex UIs with limited grasp of syntax. On the other hand, comparisons between models with higher compilation rate primarily reflected output relevance and UI design quality.

8.5.3 Performance Over Time

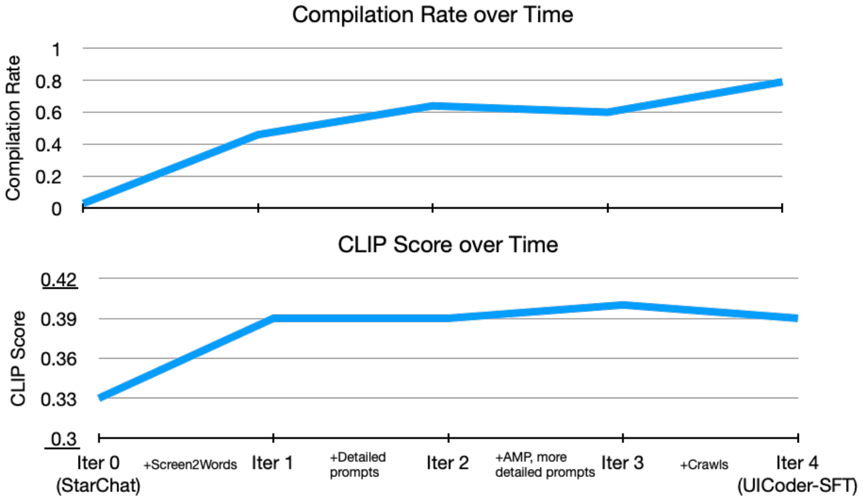


Figure 8.5: We measured model performance over time during the training process of the UICoder-SFT model. We plot two automatically calculated metrics: compilation rate and CLIP score. In general, the compilation rate and CLIP score increased with more training iterations, and the largest rate of improvement occurred during early iterations. The training process was not entirely homogeneous: we gradually incorporated different parts of our dataset between iterations (shown in the margins of the plot), which contributed to fluctuations in performance. Note that unlike Figure 8.2, metrics in this plot were calculated over the evaluation set; therefore, a different compilation rate is reported for StarChat-Beta.

We measured the impact of our training procedure by running automatic evaluations of the UICoder model after each iteration, starting from the initial StarChat-Beta base model to the fourth iteration that resulted in UICoder-SFT. The results of this experiment are shown in Figure 8.5. Overall, there is a positive trend between the number of training iterations and the performance of both metrics, and the largest rate of improvement occurs during the first iteration. The highest compilation rate was reached after the fourth iteration (0.79), while the highest CLIP Score was reached on the third iteration (0.40). Following the first iteration, we noticed that an increase in one metric sometimes led to a decrease in the other. Upon manual inspection, we noticed that more complex and longer programs were likely to have higher CLIP scores but were also more likely to contain an error. We believe that further tuning of our filters and deduplication stage could help mitigate this in the future.

Additional UI descriptions were incorporated at various points in the training process based on our manual observations and tests. For example, at the initial iteration from 0 to 1 we started with only human-authored descriptions found in Screen2Words [315]. However, by inspecting the outputs generated after the first iteration, we noticed that the model tended to produce relatively simple outputs and struggled to understand more detailed descriptions, which led us to include LLM-augmented and paraphrased descriptions. Similarly, we later included iOS screenshots, since the Android UIs in Screen2Words [315] and RICO [80] use different design patterns and UI components. These changes likely contributed to some of the fluctuations in model performance measured over the course of training.

Both the plotted metrics (Figure 8.5) and later performance measurements (Table 8.1) suggest that automated metrics have mostly converged to a maximum value. Nevertheless, it could be beneficial to continue running training iterations to take advantage of small incremental improvements or new, potentially automatically-collected sources of data [328].

8.5.4 Finetuning Other Models from Generated Data

We trained the UICoder models through a multi-iteration training process, however this is time-consuming because it requires repeatedly generating, evaluating, and training on self-generated data. Therefore, we explored the possibility of using data generated during UICoder's training to finetune other LLMs, without needing to repeat the self-generation process.

Distilled Models. Model distillation refers to the practice of using the results of a larger "teacher model" to finetune or train a smaller "student model." This practice is frequently employed to use the often superior results of proprietary models to train other more open models, but can also be used to transfer skills from a smaller purpose-built model to a larger general model through reverse-distillation. Distillation might be beneficial in our scenarios for three reasons. *i.* Given the rapid pace of base model improvements, it could be efficient to "rebase" an existing UICoder model onto a newer improved base model rather than repeat the multi-iteration training process starting with the improved base model. *ii.* Larger, general-purpose LLMs may have a higher capacity to store knowledge but are much more compute-intensive to train using our self-generation approach. It could be beneficial to transfer the specialized knowledge learned by UICoder to the general-purpose model, which might gain overall improved performance by combining each models' strengths. *iii.* Finally, UICoder data could be used to "distill" a smaller model, which might be necessary for low-compute applications. For our evalution, we chose three models to represent each of these scenarios. Octocoder is a more recent version of our base model, StarChat-Beta [233]. Octocoder was released after we began our model training, so we investigate the possibility of rebasing UICoder onto an improved model. MPT-30B-Instruct [301] is a 30B LLM that is double the size of UICoder and was finetuned on a collection of permissively-licensed general-purpose instruction-following datasets. MPT-7B-Instruct [301] is a smaller 7B version of MPT-30B-Instruct that is half the size of UICoder and might be useful for efficient deployment.

We used training examples generated from the last iteration of UICoder training to distill these models using the same hyperparameters. We refer to the resulting models as Octocoder++, MPT-30B-Instruct++, and MPT-7B-Instruct++.

Results. We used automated metrics to measure the performance of these models before and after the distillation process. Both initial and distilled model performance were highly dependent on the size of the base model, where larger model sizes led to better performance. The best-performing model was MPT-30B-Instruct which had a compilation rate of $0.14 \rightarrow 0.78$ and CLIP score of $0.351 \rightarrow 0.401$. This was followed by MPT-7B, with a compilation rate of $0.13 \rightarrow 0.69$ and a CLIP score of $0.350 \rightarrow 0.395$. Although Octocoder was a 15B model trained on code, it had the worst performance with a compilation rate of $0.06 \rightarrow 0.51$ and a CLIP score of $0.235 \rightarrow 0.382$. This may have been due to a mismatch between Octocoder's training data, which was based on scraped commit messages, and the types of UI descriptions relevant to our task. Overall, all models' compilation rates and CLIP scores were greatly improved by distillation, which suggests the utility of UICoder-generated data for finetuning other models.

8.5.5 Baseline Comparison

We compared the performance of UICoder models, distilled models, and several classes of baselines.

Baseline Models. We categorized baselines into three categories: *i)* proprietary models, *ii)* restricted models and *iii)* permissive models. We evaluated two baselines for each category.

Currently, proprietary models have the best performance, but are accessible only through web API requests and often have usage restrictions. We included GPT-4 and GPT-3.5-Turbo as proprietary baselines, since they have been shown to excel at a wide range of tasks, including code generation [52, 215]. Details about the models' architecture and training procedure are not publicly released [245]; however, GPT-3, a predecessor to both models, is known to have 175B parameters [51].

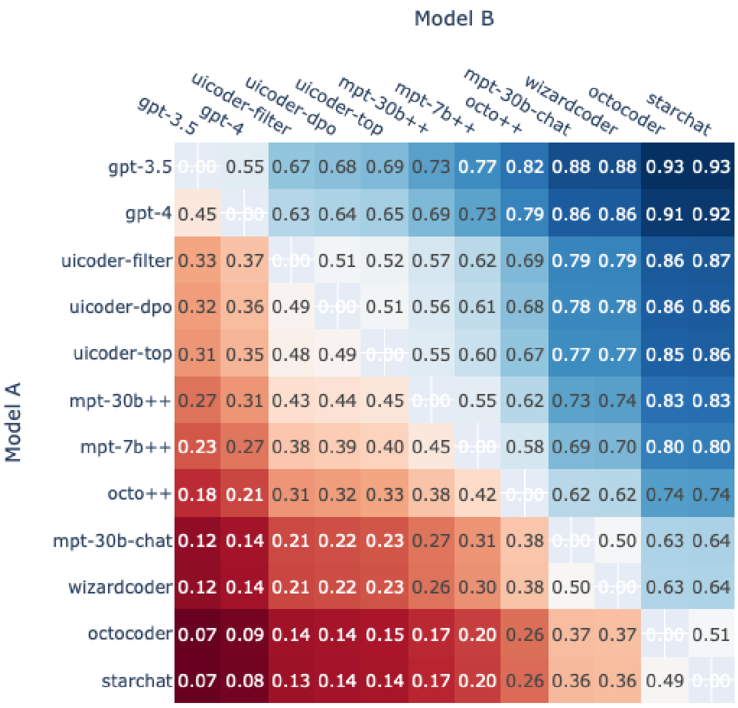
Restricted baselines are freely downloadable models with license or usage restrictions (i.e., no commercial use), due to the use of a proprietary model in generating training data. [2] We included WizardCoder [215] and MPT-30B-Chat [301] as restricted baselines. WizardCoder is a 15B model that at time of writing is the highest-performing downloadable model on the HumanEval benchmark [215], and was trained using a sophisticated distillation algorithm to query complex training examples from GPT-4. The same examples were used to finetune StarCoder [190]. MPT-30B-Chat is one of the strongest general-purpose models that can fit on a single A100 GPU at default precision [301]. Note that MPT-30B-Chat is different from MPT-30B-Instruct, in that the chat model was finetuned using output from ChatGPT and GPT-4.

Finally, we included StarChat-Beta [311] and OctoCoder [233] as permissive baselines, which were both trained on permissively-licensed code repositories and instruction-tuning datasets. StarChat-Beta is our base model, and we included OctoCoder because at the time of writing it is the best-performing permissive model on variations of the HumanEval benchmark [233]. Similar to StarChat-Beta and UICoder models, OctoCoder is also derived from StarCoder [190].

[2] Some models, such as LLaMA and LLaMA-2 are not freely downloadable because they require a license application to first be approved by Meta to access the weights.

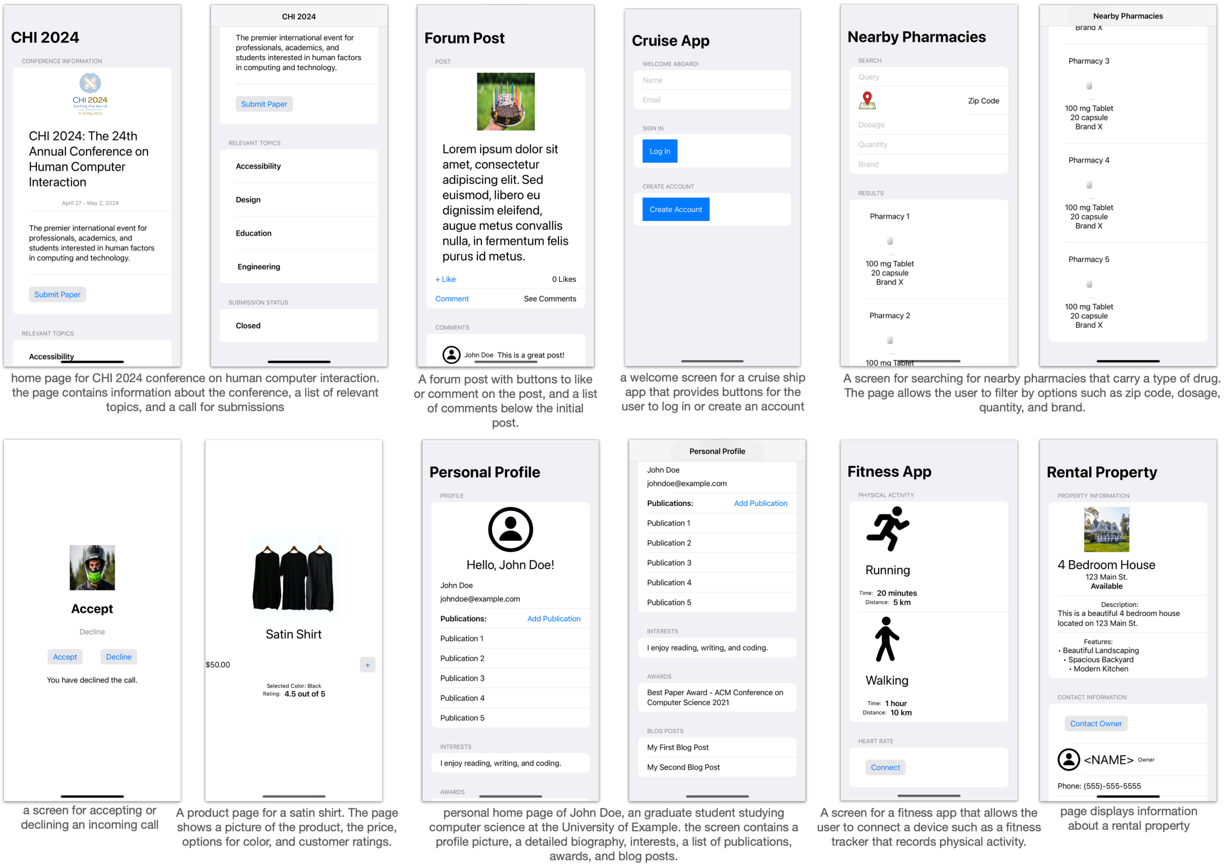
Table 8.1: Table of automated metrics and Elo ratings computed for each model on the evaluation set. Compilation rate refers to the portion of outputs that led to a compilable SwiftUI program. CLIP Score is an automatically computed estimation of quality based on the CLIP similarity score between the rendered screenshot and the original input prompt. The CLIP Score is only computed for the portion of compilable programs. The Elo rating is computed from pairwise human preference data. CLIP Scores and Elo ratings are displayed as mean ± standard deviation.

Model	Params	Compilation	CLIP Score	Elo
GPT-3.5-Turbo	-	0.88	0.416 ± 0.069	1224 ± 15.2
GPT-4	-	0.81	0.419 ± 0.070	1189 ± 15.1
WizardCoder	15.5B	0.23	0.393 ± 0.071	870 ± 12.9
MPT-30B-Chat	30B	0.27	0.368 ± 0.073	873 ± 12.9
StarChat-Beta	15.5B	0.03	0.334 ± 0.041	773 ± 10.2
Octocoder	15.5B	0.06	0.235 ± 0.068	777 ± 10.5
UICoder-Filtered	15.5B	0.79	0.404 ± 0.063	1099 ± 15.2
UICoder-Top	15.5B	0.82	0.396 ± 0.061	1084 ± 15.7
UICoder-DPO	15.5B	0.75	0.393 ± 0.060	1091 ± 15.6
MPT-7B-Instruct++	7B	0.69	0.395 ± 0.064	1015 ± 14.4
Octocoder++	15.5B	0.51	0.382 ± 0.061	959 ± 14.8
MPT-30B-Instruct++	30B	0.78	0.401 ± 0.063	1047 ± 15.0



	gpt-3.5	gpt-4	uicoder-filter	uicoder-dpo	uicoder-top	mpt-30b++	mpt-7b++	octo++	mpt-30b-chat	wizardcoder	octocoder	starchat
gpt-3.5	0.00	0.55	0.67	0.68	0.69	0.73	0.77	0.82	0.88	0.88	0.93	0.93
gpt-4	0.45	0.00	0.63	0.64	0.65	0.69	0.73	0.79	0.86	0.86	0.91	0.92
uicoder-filter	0.33	0.37	0.00	0.51	0.52	0.57	0.62	0.69	0.79	0.79	0.86	0.87
uicoder-dpo	0.32	0.36	0.49	0.00	0.51	0.56	0.61	0.68	0.78	0.78	0.86	0.86
uicoder-top	0.31	0.35	0.48	0.49	0.00	0.55	0.60	0.67	0.77	0.77	0.85	0.86
mpt-30b++	0.27	0.31	0.43	0.44	0.45	0.00	0.55	0.62	0.73	0.74	0.83	0.83
mpt-7b++	0.23	0.27	0.38	0.39	0.40	0.45	0.00	0.58	0.69	0.70	0.80	0.80
octo++	0.18	0.21	0.31	0.32	0.33	0.38	0.42	0.00	0.62	0.62	0.74	0.74
mpt-30b-chat	0.12	0.14	0.21	0.22	0.23	0.27	0.31	0.38	0.00	0.50	0.63	0.64
wizardcoder	0.12	0.14	0.21	0.22	0.23	0.26	0.30	0.38	0.50	0.00	0.63	0.64
octocoder	0.07	0.09	0.14	0.14	0.15	0.17	0.20	0.26	0.37	0.37	0.00	0.51
starchat	0.07	0.08	0.13	0.14	0.14	0.17	0.20	0.26	0.36	0.36	0.49	0.00

Figure 8.6: Matrix shows the predicted win probability of model A against model B.



home page for CHI 2024 conference on human computer interaction. the page contains information about the conference, a list of relevant topics, and a call for submissions

A forum post with buttons to like or comment on the post, and a list of comments below the initial post.

a welcome screen for a cruise ship app that provides buttons for the user to log in or create an account

A screen for searching for nearby pharmacies that carry a type of drug. The page allows the user to filter by options such as zip code, dosage, quantity, and brand.

a screen for accepting or declining an incoming call

A product page for a satin shirt. The page shows a picture of the product, the price, options for color, and customer ratings.

personal home page of John Doe, an graduate student studying computer science at the University of Example. the screen contains a profile picture, a detailed biography, interests, a list of publications, awards, and blog posts.

A screen for a fitness app that allows the user to connect a device such as a fitness tracker that records physical activity.

page displays information about a rental property

Figure 8.7: Screenshots rendered from SwiftUI code generated by our UICoder models. Overall, the generated UIs follow the original description; however, there are several instances where part of the input was ignored. Note that during the evaluation study, all images were replaced with the same placeholder icon, but for illustration purposes we manually included stock photos and icons. The model-generated code used to render these screenshots were not modified in any way except to update image asset names, but we generated multiple outputs for each input description and chose the best one through manual inspection.

Results. We ran all three metrics for the baseline comparison experiment. The results are summarized in Table 8.1 and Figure 8.6 shows the expected human ratings for every pair of evaluated models. Overall, the proprietary models had the best performance, followed by UICoder models, then models distilled from UICoder data. GPT-3.5-Turbo had the highest compilation rate and the highest Elo score, while GPT-4 had the highest CLIP Score. It was somewhat surprising that GPT-3.5 had a higher compilation rate than GPT-4, since GPT-4 is often considered a stronger model. Upon manual inspection, we hypothesized that GPT-4 often tried to produce longer and more complex code implementations, which made mistakes slightly more likely. A lower compilation rate also affected the Elo score, since uncompilable programs were automatically counted as losses in the pairwise comparisons. The higher CLIP score suggests that the compilable GPT-4 programs had high visual relevance to the input prompt.

Variations of the UICoder models approached the performance of the proprietary GPT models, and UICoder-Top had a higher compilation rate than GPT-4. All three UICoder variants had roughly the same level of performance, which suggests that the additional preference alignment stage did not lead to significant improvements, possibly due to sub-optimal hyperparameters or the additional model quantization needed to run the training algorithm.

Distilled models finetuned from the final iteration of generated data had slightly lower performance than the UICoder models themselves, which together with our performance over time experiments (Figure 8.5) suggests that multiple training iterations are needed to maximize performance. MPT-30B-Instruct++ was the best-performing distilled model, possibly due to its larger model size, and although Octocoder++ is pre-trained in large amounts of code, it performed worst out of the distilled models. MPT-7B-Instruct++ maintains relatively high performance with a much smaller size, which is encouraging from an efficiency standpoint.

All other downloadable models fared considerably worse. Most of the programs generated by these models failed to compile due to calling undefined variables or functions. Restrictive models performed better than permissive ones, due to the use of larger amounts of finetuning data generated from proprietary model APIs. It is possible that the distilled proprietary data did not contain many Swift-related coding examples and querying proprietary models specifically for more UI-related tasks could further boost performance. Notably, StarCoder-Beta, our base model, was ranked in last place, likely due to the scarcity of Swift and SwiftUI code in its training code. Our results suggest that our method is highly effective at improving its capabilities, since after training, UICoder has became one of the top performing models.

8.6 Discussion & Future Work

Much current work on code generation focuses on code that can be validated purely with unit testing, as demonstrated by the widely used HumanEval benchmark [65]. User interface code is often quite different from unit testable code, however, as UI code can rarely be described in terms of defined input/output pairs or the output of a mathematical function. UI code has a lifecycle that is longer than a single function call, must react to human events, and often relies on "back-end" technologies to populate its data. Furthermore, UI code often defines a visual design, for which there are many technically "correct" alternatives and the quality of any particular alternative may be influenced by the tastes of the user who views it and the context in which it is shown, such as on a mobile device vs. a traditional desktop computer. Thus, we believe that generating high-quality UI code is a substantial challenge beyond typical code generation tasks.

Another challenge for UI code generation is that this problem cannot be solved with just the knowledge of a programming language, such as Python or Swift. UIs are produced by toolkits, which are APIs that extend the language with specific functionality for user interfaces. There are many UI toolkits, each with its own features and peculiarities. Even if a programming language is well represented in the training set for an LLM, any particular UI toolkit for that language will almost certainly be less represented. Languages with poor representation in training sets, such as Swift, will have UI toolkits with even less representation.

These aspects demonstrate the challenge faced by UICoder. Not only was it required to generate user interface code, which is already hard, but it faced the challenge of doing so with a language and API (Swift and SwiftUI) that are poorly represented in LLM training sets. Thus, we think it is particularly impressive that the method presented here was able to boost performance on compilability and visual relevance to levels that are nearly competitive with the best available proprietary LLMs. Furthermore, our method was able to boost performance without the introduction of any new human-generated code, which can be very expensive to collect and curate.

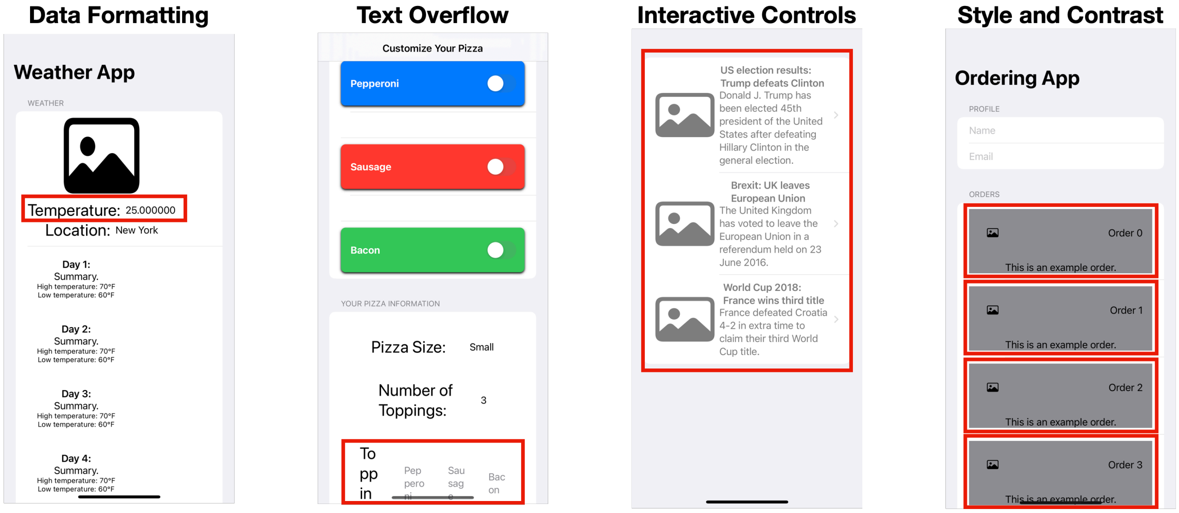
While we focused specifically on Swift and SwiftUI, our method is not specific to either the language or the API that is used. Further experimentation is required, but we believe our method should be applicable to other languages and their UI toolkits, even if they have a low representation in standard code training sets. The compilability aspect of our approach might also be applicable to teaching code generation models how to use other kinds of non-UI APIs with low training set representation.

8.6.1 Limitations of Automated Feedback

The goal of our training object is to maximize the automated scores provided by a compiler and the CLIP vision-language model. Although our experiments (Figure 8.5) suggest that we have mostly achieved this goal by "saturating" these metrics, usage-based testing and collected human ratings indicate there is still room for improvement. In this section, we discuss the limitations of our currently chosen feedback mechanisms and discuss opportunities for improvement.

Swift Compiler. The Swift compiler provides ground truth for compilation success, but this only returns a sole binary value, which equates the quality of any two non-compiling programs regardless of the number of defects. Based on our observations (Figure 8.2), this incentivizes the model to avoid errors, rather than learning how to correct them, resulting in very simple outputs. We attempted to rank generated programs by the proportion of error-free lines of code to total program length [207] (Section 8.3.4) but found that this actually led to decreased a compilation rate in the case of the UICoder-DPO model. A more useful signal would be to count the number of changes needed to "fix" a failing program, but this cannot be directly computed from compiler output (e.g., correcting a typo in a single method definition could resolve multiple errors). Thus, we believe an important future direction is incorporating PL or verification techniques as a better source of automated feedback for language syntax.

CLIP Score. Compared to the discrete binary output provided by the compiler, the CLIP score provides a finer-grain numerical output. We found that CLIP could detect overall output relevance and large disparities in output quality (e.g., eliminating "obviously" bad designs), but it wasn't well-equipped for comparing subtle design choices or detecting some types of visual defects. Figure 8.8 shows our trained model still exhibits errors that are uncharacteristic of human developers, including data formatting, text-overflow, inaccessible controls, poor style, and contrast. We see three avenues for improving CLIP's performance. First, we hypothesize that a major reason for CLIP's limitations is the small fraction of its training examples that included UI screenshots. One promising approach to improve CLIP's performance is to fine-tune the CLIP model using datasets specific to user interfaces, such as Screen2Words [315], or even fine-tune on a subset of high-quality text-image pairs generated by UICoder. The CLIP model is also known to struggle with certain tasks such as object-counting and reasoning [260], which could be relevant for some types of prompts, e.g., "a login page with three buttons stacked vertically." Integrating additional models, such as those that can explicitly detect UI elements [347] and



US election results: Trump defeats Clinton
Donald J. Trump has been elected 45th president of the United States after defeating Hillary Clinton in the general election.

Brexit: UK leaves European Union
The United Kingdom has voted to leave the European Union in a referendum held on 23 June 2016.

World Cup 2018: France wins third title
France defeated Croatia 4-2 in extra time to claim their third World Cup title.

This is an example order.

This is an example order.

This is an example order.

This is an example order.

“A weather app that shows a map of a location and a daily weather forecast.”

"page for customizing a pizza order on the Pizza Hut app"

“The screen displays videos related to recent news top stories”

“The profile screen from an ordering app, featuring many features to inspect current and previous orders, maintain shopping lists and manage coupons.”

Figure 8.8: We show four types of failure cases observed in generated data. Data formatting errors occur when a sub-optimal method is used to represent data (e.g., numbers) as text or other widgets. Here, the temperature portion of the weather app is shown with unrealistically high decimal precision. Text overflow errors occur when the model makes text too big for its container, which causes it to overflow onto multiple lines. Some interactive controls generated by the model are not constructed using the correct container, making them not tappable during actual use. Finally, the model can make sub-optimal styling decisions that make text hard to read due to low contrast. Note that all icons and images in these samples were replaced with placeholders.

structure [326], or tools that detect style and accessibility-related errors [28], could help address the shortcomings of the CLIP model. Finally, because CLIP only accepts image input, it can only evaluate a dynamic, interactive UI with a screenshot of its default state. Integrating crawler-based approaches [204, 328] that can directly interact with UI affordances could provide a more holistic evaluation of UI quality.

8.6.2 Going Further with Human Guidance

We focused on fully automated methods for training LLMs to generate SwiftUI code. There are cost and efficiency benefits to this approach, however we found that automated metrics alone are insufficient for measuring UI quality. A natural question is: “What are opportunities for humans to further improve the process?” In this section, we discuss opportunities for efficiently incorporating human guidance during training and evaluation.

Improved Training. Previously, human annotators provided guidance during the training process by *i)* creating labeled examples for the supervised finetuning stage and *ii)* providing preference ratings during the model alignment stage. Authoring labeled examples is an expensive and time-consuming process that requires finding, training, and compensating skilled developers to write many SwiftUI code examples. Our results suggest that the UICoder-SFT model, trained with self-generated labeled data is an effective substitute. Our method falls short when generating preference rankings needed for alignment training, due to scoring imperfections from our automated measurements. In general, rating UIs is much easier than authoring them, since the process is much faster and does not require coding knowledge to complete. In future work, we plan to explore preference-based human feedback and test the improvement of human-provided rankings over automatically generated ones.

Efficient Evaluation. We did use human preference rankings to evaluate and compare model performance using the Elo rating system to estimate the relative performance of models. While the Elo score seems to be a more reliable indicator of UI quality than automated methods such as CLIP, one drawback is that it is time-consuming to run the evaluation and the Elo ratings may be difficult to reproduce because they are dependent on the pool of human raters. An interesting future direction would be to use human ratings to first train a separate model to predict human ratings [100], then use the resulting model as a more reliable automated evaluator in place of CLIP. Similar to the "reward model" used by RLHF techniques [250], such a model would facilitate more efficient evaluation through human-calibrated data.

A complementary direction would be to pursue opportunities for richer human evaluation, beyond simple preference ranking. For example, a human rater could provide feedback such as "Don't use grey backgrounds when displaying black text" (Figure 8.8), which could automatically adjust the scores of multiple samples that contain the described characteristic. Because all screenshots are rendered directly from code, it is also possible to map image regions directly to locations in the code, raising opportunities for other types of fine-grained feedback interactions, e.g., a human might mark a badly placed UI element and then feedback could be given to the model about the code that placed this element.

8.7 Conclusion

In this paper, we introduced a novel method that uses automated tools such as a code compiler and a pre-trained vision-language model to finetune LLMs to generate UI code from user-provided textual descriptions. Unlike other instruction-following LLMs that are trained on expensive human feedback or output from a stronger proprietary model, our technique trains LLMs entirely using high-quality examples mined from self-generated data. We applied five iterations of our algorithm to an existing open-source LLM, resulting in nearly one million generated SwiftUI programs, which were then filtered and refined to train the UICoder model. In a series of experiments that measured both automated metrics and human preferences, UICoder and other models trained with our method outperformed all other downloadable baselines by a large margin and approached the performance of larger proprietary models.

9

UIClip: A Data-driven Model for Assessing User Interface Design

9.1 Preamble

Content in this chapter is based on a pre-print that is under submission at a conference [331].

The UICoder system described in Chapter 8 presented an alternative to Reflow's layout optimization approach 7 by first generating declarative UI code that could be compiled into a fully interactive application. Both approaches had trade-offs. Reflow could re-use many parts of the original rendered UI, which could lead to more consistent user experience, and its formulation lent itself more naturally to certain types of objectives (*e.g.,* spatially-dependent interaction). On the other hand, UICoder's code generation approach had advantages such as theoretically supporting more substantial changes, but it also required the model to also understand learn the syntax of the UI programming language, which introduced additional complexity. In both cases, however, the resulting UIs often contained design defects, such as mis-aligned UI elements (in Reflow) or poor layout and color choice (in UICoder). This is due, in part, to the lack of an automated assessment method that can identify and penalize the presence of bad design choices while encouraging ones. To this end, this chapter introduces UIClip, a data-driven model for assessing the quality of UI designs. I show that UIClip can be used to improve these automated UI generation approaches by scoring output quality and integrated into design tools for human designers.

UI design is a difficult yet important task for ensuring the usability, accessibility, and aesthetic qualities of applications. In our paper, we develop a machine-learned model, UIClip, for assessing the design quality and visual relevance of a UI given its screenshot and natural language description. To train UIClip, we used a combination of automated crawling, synthetic augmentation, and human ratings to construct a large-scale dataset of UIs, collated by description and ranked by design quality. Through training on the dataset, UIClip implicitly learns properties of good and bad designs by i) assigning a numerical score that represents a UI design's relevance and quality and ii) providing design suggestions. In an evaluation that compared the outputs of UIClip and other baselines to UIs rated by 12 human designers, we found that UIClip achieved the highest agreement with ground-truth rankings. Finally, we present three example applications that demonstrate how UIClip can facilitate downstream applications that rely on instantaneous assessment of UI design quality: i) UI code generation, ii) UI design tips generation, and iii) quality-aware UI example search.

9.2 Introduction

What makes a good user interface (UI)? It is hard to comprehensively articulate what separates a good UI design from a bad one, and the task of UI design is challenging even for experts with years of training and practice. Guidelines exist that list some general principles [241, 285], but they are often insufficient or difficult to operationalize, especially for novice designers. Because of this, many application UIs today contain common design problems, which can negatively impact usability, accessibility, and design aesthetics.

The most holistic method of evaluating UIs is *usability testing*, which can uncover UI design flaws, accessibility problems, and software bugs, but it is generally a time-consuming and costly process. Approximate assessments, such as *heuristic evaluation*, rely on experts applying a set of pre-defined principles to rapidly identify potential problems and estimate overall UI quality. However, even these abbreviated strategies can be difficult to employ consistently or in the absence of a knowledgeable expert.

To this end, computational methods have been developed to estimate the quality of UIs, taking into account factors such as visual aesthetics [227], cognitive principles [246], and context [249]. Because of their automated nature, they unlock new opportunities for UI design [297, 306] and evaluation [230]. However, most of these prior computational approaches are limited. Some techniques apply objectives and metrics inspired by cognitive principles [227, 246], such as visual complexity, layout quality, and color harmony to UI designs, but their outputs still require interpretation and cannot, for example, be used to compare the quality of two candidate designs. Other approaches are toolkits that learn user-specific models for generating adaptive interfaces [110, 111, 112], and they also cannot be applied to more generalized UI design tasks.

Our paper introduces a novel computational model, *UIClip*, to assess the design quality of any UI from its screenshot. UIClip is based on the well-known CLIP vision-language model [260], and it uses a natural language description of the UI coupled with a screenshot to assign a numerical score that estimates design quality. CLIP, by default, is not well-suited for judging UI quality and relevance. Therefore, to train UIClip, we developed a novel technique for synthetically generating a large-scale dataset of UIs ranked by design quality. Our strategy takes existing UIs (e.g., web pages) and intentionally introduces design defects by modifying style and layout attributes. The process created pairs of original and "jittered" interfaces and allowed the models to learn the differentiation between these pairs. We used this method to generate 2.3 million pairs of UIs coupled with their quality-related descriptions. To align our model with real-world design preferences, we collected 1.2K ratings from professional designers on an extra UI set. These

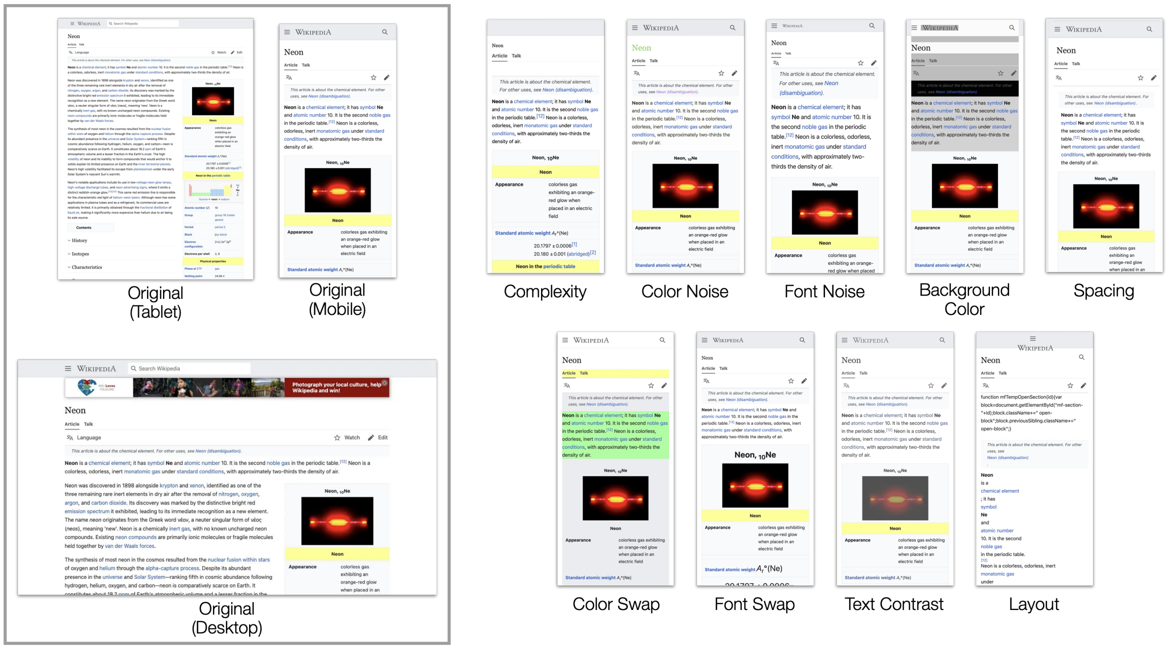


Figure 9.1: Our synthetic dataset was comprised of UIs that were processed by *jitter functions* to introduce design defects. In this figure, we visualize the effect of each jitter function independently, although up to three jitter functions can be applied simultaneously. Our crawler captures screenshots for multiple devices (desktop, tablet, and mobile), but due to space constraints, we only show rendered mobile examples.

ratings were used to refine UIClip and validate the effectiveness of our model.

We benchmarked UIClip with other large vision-language models (LVLM) by evaluating them on a held-out set of UI screens. We assess the models on three tasks, including design quality, improvement suggestions, and design relevance. The results showed that UIClip outperformed all other models in every task, despite being smaller in size. Finally, to demonstrate the utility of UIClip, we present three example applications that use our model to provide different types of computational UI design assistance: *i)* quality-aware UI code generation, *ii)* UI design suggestion generation, and *iii)* quality-aware UI example retrieval.

To summarize, our work makes the following contributions:

1. A large-scale dataset of UI designs and descriptions comprised of synthetic and human-generated design ratings.
2. A computational model that scores UI screenshots based on relevance to a textual description and design quality.
3. Three example applications that demonstrate how UIClip can be used to facilitate downstream applications: *i)* a tool that improves the quality of UI code generated by LLMs, *ii)* a tool that generates design recommendations for a UI screenshot, and *iii)* a UI design search engine.

9.3 Datasets for UI Design Quality

While several UI datasets exist, they are annotated for other applications, such as element detection [53, 80], natural language description [315], and app categorization [181]. Although some prior work has rated model-generated UI code [111, 286], to our knowledge, no publicly available, large-scale dataset exists for UI design assessment. To this end, we collected over 2.3 million UI screenshots, each paired with natural language text that includes a caption, design quality, and design defects. Since it is prohibitively costly and time-consuming to collect enough human-annotated data to train deep learning models, the majority of our data (over 99.9%) is synthetically generated, and a small set of human ratings is collected from designers. We refer to our synthetically-generated dataset as JITTERWEB and our human-rated dataset as BETTERAPP.

9.3.1 Synthetic Data

JITTERWEB is a synthetic dataset of 2.3 million examples created through automated web crawling, data augmentation, and captioning. Recent research has shown that UIs on the web (*e.g.,* web pages), are a useful source of data for data-driven UI modeling, due to the relative ease of applying automated crawling techniques and extracting semantic metadata from the browser [170, 330]. The main idea behind our synthetic data approach was to first visit an existing web page and record its appearance (*i.e.,* take a screenshot), then randomly apply several *jitter functions* that intentionally degrade the design quality of the web page in different, controllable ways and record the resulting appearances. Jitter functions are implemented as snippets of JavaScript code that, for example, add random noise to CSS attributes or swap colors in the web page's color palette. The result of applying this process to a web page is one "original" sample paired with several variations of itself, each with a set of known design defects. Through this process, we are able to construct a large-scale dataset to learn the *relative design quality* of UIs. In other words, we assume that the original UI is mostly better than the jittered one. Instances where the jittered version outperforms the original are uncommon and the noises introduced from such instances should be negligible given the scale of our dataset.

Data Collection

We followed the collection methodology of WebUI [330], where a headless Chrome browser was used to visit thousands of websites with different simulated client devices (*e.g.,* mobile phone, desktop, tablet). It was not possible to directly re-use the publicly-released WebUI data, which consists of screenshots and extracted metadata, because our data augmentation pipeline necessitates loading the website in a browser to run the *jitter functions*, which are implemented as JavaScript code. Unlike the crawler used in WebUI, we adopted a simpler architecture that directly crawls URLs from publicly available datasets. We crawled nearly 300,000 web pages, using URLs from the MC4 dataset provided by the Allen Institute for AI [88], which is an adaptation of the original C4 dataset [262] frequently used to train large language models [43, 71, 178, 308]. This dataset has undergone screening to remove explicit content [88]. In addition, we excluded URLs that resulted in 404 errors.

JitterWeb was randomly partitioned into training (80%), validation (10%), and test (10%) splits by web page URL. We further randomly selected 201 samples from the original test split, to make it the same size as the test split from our human-rated data (BetterApp) for model evaluation.

Jitter Functions

Jitter functions are JavaScript code snippets that are used to controllably introduce design defects into web pages. To design these functions, we reviewed various guidelines on usability and design evaluation found in design textbooks [206, 285], online resources [123, 323], and published literature [216]. While undoubtedly useful for informing application design, many of the principles described in these resources could not be assessed by looking at a single screenshot (*e.g., "error prevention," "user control and freedom").* We ultimately chose the *CRAP* guidelines [321], which are four general principles for UI visual design relevant to our task: contrast, repetition, alignment, and proximity. We developed the jitter functions based on a combination of these guidelines and what is possible to programmatically adjust through JavaScript and CSS styling.

Below, we describe the functions that we implemented and the CRAP principles that inspired them:

- Colors
    - Color Swap (contrast, repetition) - Randomly swaps the colors of elements on the web page
    - Color Noise (contrast, repetition) - Adds numerical noise to CSS attributes for RGB values
- Font
    - Font Size (contrast, repetition) - Randomly swaps the font sizes of text elements in the page (*e.g.,* swapping the size of subheading text with the size of body text)
    - Text Noise (contrast, repetition) - Adds numerical noise to CSS attributes for text size
- Contrast
    - Text Color (contrast) - The contrast of text is decreased so that it appears closer to its container's color
    - Background Color (contrast) - Makes the background color of containers containing text closer to the color of the text.
- Spacing (alignment, proximity) - Adds numerical noise to CSS attributes for margin and padding
- Complexity (contrast, repetition, alignment, proximity) - Randomly removes images, text, and other element styling
- Layout (alignment, proximity) - Modifies CSS related to element layout such as flow (*e.g.,* horizontal or vertical).

The jitter functions are composable, and when the crawler visits a web page, it chooses up to three functions via uniform random sampling to apply sequentially before taking a screenshot of the jittered UI. Figure 9.1 shows an example of a web page processed by each of our jitter functions.

Description Generation

Each UI screenshot was associated with a natural language description that includes a caption, design quality, and a list of design defects (inferred from the applied jitter functions). The full description is formatted by concatenating multiple components: *i)* a constant prefix ("ui screenshot."), *ii)* a design quality tag ("poor design" if the screen has been jittered, otherwise "well-designed"), *iii)* a list of design defects (*e.g.,* if the "text contrast" jitter function was applied, a suggestion would be "bad text contrast"), and *iv)* a caption describing the screenshot. Figure 9.2 provides a visual illustration of this process.

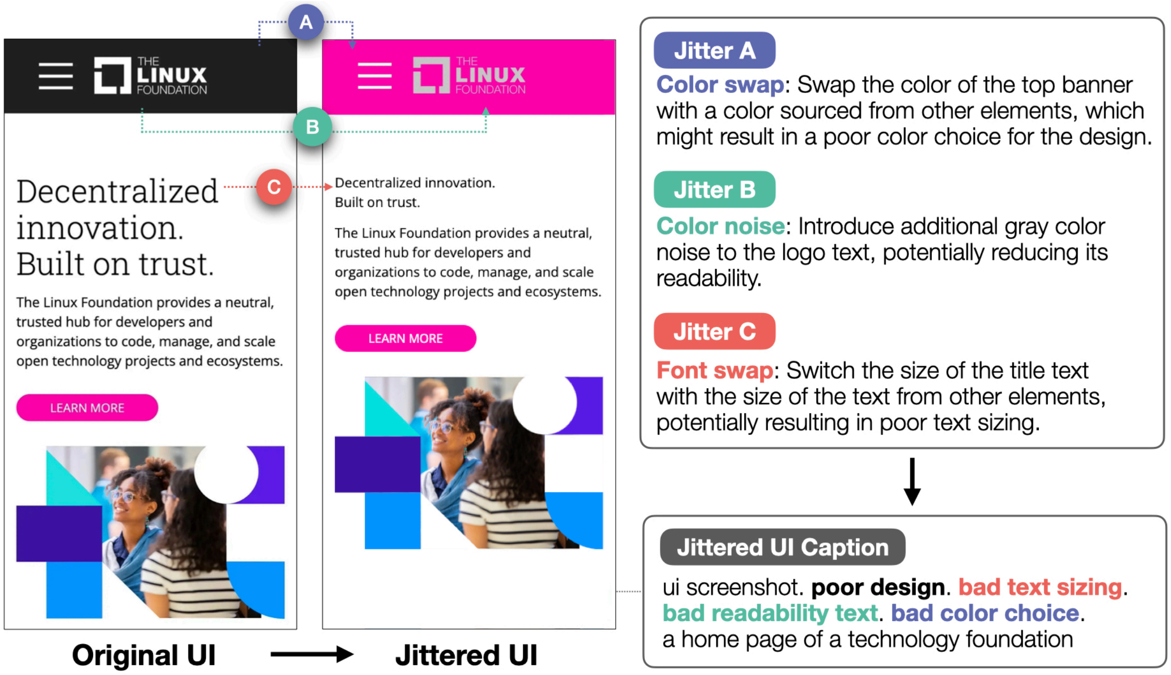
The design-related components are inferred from the jittering process. To generate the caption, we used a set of pre-trained models to predict [180, 315], then paraphrase [155] a caption from the UI screenshot. In the generation process, we specifically avoided the use of models with restrictive usage agreements or trained using data from models with restrictive usage agreements [1]. Because the introduction of design defects by jitter functions may affect the accuracy of the captioning model, we generate the caption for each original UI, and then propagate the caption to all its variations.

9.3.2 Human-rated Data

While our synthetic approach to automatically generating pairs of design preferences can be efficiently scaled to millions of screenshots, it also has drawbacks. In the synthetic dataset, preferences are only generated between variations of the same screen, which does not reflect comparison between independent designs. While we used established design principles to author jitter functions, they may not represent the actual distribution of design flaws across real-world apps, *e.g.,* small element margins may be a very common problem "in-the-wild" but is only represented in one of our heuristics. Finally, a part of the creation process for the synthetic dataset involves using a pre-trained UI screenshot captioning model for caption generation. This model may produce incorrect captions that limit a downstream model's ability to understand UI design relevance. To this end, we collected the BETTERAPP dataset using feedback from human designers. BETTERAPP addresses the drawbacks of synthetic data by *i)* comparing UI screens from different apps, *ii)* collecting design defects from real apps, and *iii)* using human-improved UI captions.

As a starting point, we used an existing public dataset called VINS [53], which contains screenshots of iOS apps, Android apps, design mockups, and lower-fidelity design artifacts such as wireframes. Because it was originally used for design search and element detection applications, the VINS dataset contains screenshot images and element annotations. For our application, we only use the screenshot images and not the lower-fidelity wireframes. In addition to VINS data, we also included screenshots of UIs rendered by an open large-language model

[1] The terms of service of proprietary model providers such as OpenAI, Llama, and Claude prohibit using their model outputs to train other models. Therefore we avoid them and also other "distilled" models trained on their output.


A
B
C
THE LINUX FOUNDATION
Decentralized innovation. Built on trust.
The Linux Foundation provides a neutral, trusted hub for developers and organizations to code, manage, and scale open technology projects and ecosystems.
LEARN MORE
THE LINUX FOUNDATION
Decentralized innovation. Built on trust.
The Linux Foundation provides a neutral, trusted hub for developers and organizations to code, manage, and scale open technology projects and ecosystems.
LEARN MORE
Original UI
Jittered UI

Jitter A
Color swap: Swap the color of the top banner with a color sourced from other elements, which might result in a poor color choice for the design.

Jitter B
Color noise: Introduce additional gray color noise to the logo text, potentially reducing its readability.

Jitter C
Font swap: Switch the size of the title text with the size of the text from other elements, potentially resulting in poor text sizing.

Jittered UI Caption
ui screenshot. **poor design**. **bad text sizing**. **bad readability text**. **bad color choice**.
a home page of a technology foundation

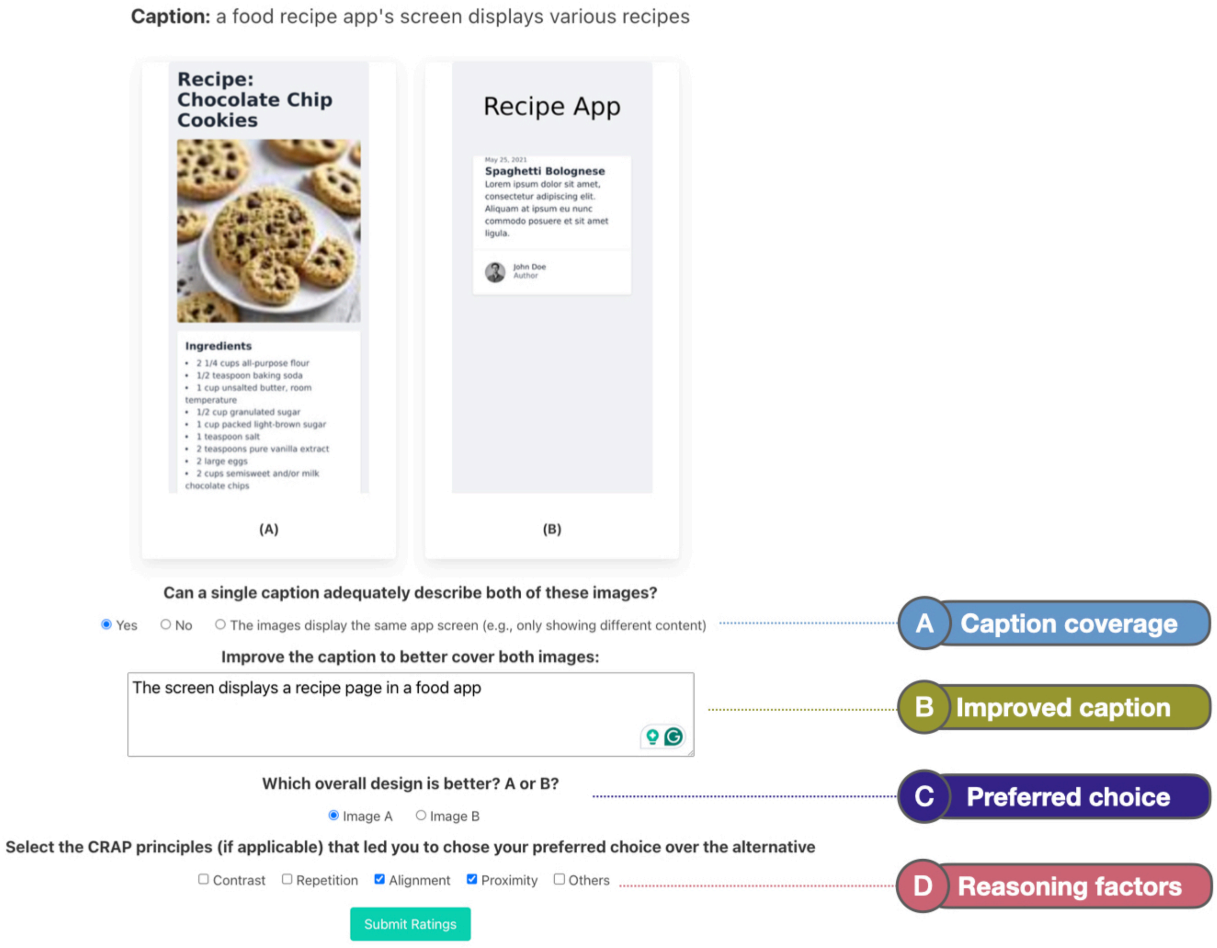
Figure 9.2: Our process for generating text descriptions for JITTERWEB. Based on a set of randomly-chosen jitter functions, several design defects are introduced, *e.g.,* color swap, color noise, font swap. These design defects are recorded as a part of the jittered UI's caption, which helps our model associate design defects with the UI screenshot.

[156] prompted to generate HTML code given natural language descriptions in our dataset. We hypothesized that these samples would contain more variation in design quality and more design defects, which could be useful for learning design quality.

To prepare the data for our rating procedure, we applied several additional processing steps. We first applied the same automated captioning model [180] used to construct synthetic examples to assign an initial caption to each dataset in VINS. These captions were later improved by participants. We used a pre-trained sentence embedding model [264] to generate a fixed-size embedding for each screen based on its auto-generated caption. Finally, we applied the DBSCAN clustering algorithm [94] to group together screenshots with similar captions. As a result of this process, the screenshots are collated so that screens of similar functionality can be found in the same cluster (*e.g.,* all login screens). We clustered the VINS and synthetic examples separately, so clusters are only made entirely of either real-world or synthetic UIs. Designers were then asked through pairwise comparisons to assign relative rankings between UI screens in the same cluster.

Designer Rating Procedure

We recruited 12 designers (ages 20-32, 11 female and 1 male) as participants at a university with varying levels of experience through word of mouth. The participants had varying backgrounds. Some had up to 8 years of industry experience in UI/UX design. Others had more informal



Caption: a food recipe app's screen displays various recipes

Recipe: Chocolate Chip Cookies

Ingredients

- 2 1/4 cups all-purpose flour
- 1/2 teaspoon baking soda
- 1 cup unsalted butter, room temperature
- 1/2 cup granulated sugar
- 1 cup packed light-brown sugar
- 1 teaspoon salt
- 2 teaspoons pure vanilla extract
- 2 large eggs
- 2 cups semisweet and/or milk chocolate chips

(A)

Recipe App

May 25, 2021

Spaghetti Bolognese

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam at ipsum eu nunc commodo posuere et sit amet ligula.

John Doe
Author

(B)

Can a single caption adequately describe both of these images?

● Yes ○ No ○ The images display the same app screen (e.g., only showing different content)

Improve the caption to better cover both images:

The screen displays a recipe page in a food app

Which overall design is better? A or B?

● Image A ○ Image B

Select the CRAP principles (if applicable) that led you to chose your preferred choice over the alternative

☐ Contrast ☐ Repetition ☑ Alignment ☑ Proximity ☐ Others

Figure 9.3: A screenshot of the application used for collecting human design ratings. Participants first decide whether the pair of screenshots can be described by a single caption (A). If possible, an improved caption is authored (B). Participants select one option that better matches the caption (C) and provide their reasons for doing so (D).

experience, but all were enrolled or had taken graduate-level courses focused on the design and implementation of UIs. Participants spent around 1.5 hours rating UI screenshots, with the goal of reaching at least 100 screenshots. Participants were compensated $10 per hour (rounded up) for their time.

Participants were first asked to review an online resource that describes and provides examples of the CRAP visual design principles [159]. Participant ratings were collected using a custom-built web application (Figure 9.3). The start page of the application displayed instructions and recorded a visitor ID, which allowed analysis of rating consistency.

Following the start page, the web application repeatedly *i)* selects a random cluster from the processed data then *ii)* randomly selects two UIs from within the cluster to display. The participant was then asked to do the following steps:

1. Write a short, one-sentence caption that contains enough detail to describe both screenshots. If one of the screenshots is irrelevant (*e.g.,* due to clustering error), write a caption

for the first screenshot.

2. Provide a relative ranking between the two screenshots given the options “A is better” or “B is better.”
3. Select all relevant CRAP principles that were important in determining the ranking, unless “about the same” was selected in the prior step.

In total, we collected around 1200 ratings from all participants. We ignored pairs that could not be described by a single caption, which led to 892 rating pairs. To measure inter-rater reliability (IRR), we initially had each participant evaluate the same set of 10 predetermined pairs. Afterward, the rating pairs were distributed randomly. We used this initial set to compute Krippendorff’s alpha score, with $\alpha = 0.37$. We discuss the factors influencing these ratings in Section 7.2, attributing the variation to the task’s inherent subjectivity and variable individual preferences, such as familiarity with Android or iOS apps.

Similar to our synthetic generation approach, responses from each step are used to construct different parts of each UI screenshot’s text description. The human-authored or human-refined caption from step 1 is used to improve the original auto-generated one. The relative ranking from step 2 is used to infer the correct design-quality tag, where the preferred example is assigned “well-designed” and the other is assigned “poor design.” If it was indicated that the two screenshots had the same quality, the design quality tag was omitted from the full description. The selected principles from step 3 are used to construct a set of design defects for the non-preferred screenshot. For example, if a participant selected the *contrast* principle as a reason for choosing A over B, then “bad contrast” is added to the generated description of B. Note that the same screenshot can appear in more than one randomly sampled pair, which could result in conflicting descriptions *e.g.,* if it was preferred in one round but not in another. Our training algorithm is robust to these collisions and over time learns to approximate a score based on the proportion of times it was preferred.

To generate BETTERAPP training (70%), validation (10%), and test (20%) splits, we randomly partitioned the UI clusters, which ensured that both UI screenshots from rated pairs always occurred within the same split. We chose the split percentages for BETTERAPP so that the size of the test set is roughly equivalent in size to other popular model benchmarks [65]. The final sizes of the splits were: train (618 pairs), validation (73 pairs), and test (201 pairs).

9.4 UIClip

We used the JITTERWEB and BETTERAPP datasets to train a computational model UIClip, that assesses UI designs from screenshots. While our datasets could be applied to train any model, such as large vision-language models [33, 211] that typically include the language decoder from an LLM, we adopted the CLIP architecture [260] as it is designed to produce a numerical score, which is similar to our objective of scoring designs. In addition, we also found this model to be more versatile for supporting a set of example applications (*e.g.,* example retrieval and scoring) and much more efficient for training and inferencing (due to much smaller size). There are several variations of the CLIP model, and we chose the smallest variation released by OpenAI called *CLIP B/32*, which contains 151 million parameters.

CLIP B/32 is a dual-encoder transformer model (*i.e.,* consisting of an image and text encoder) that accepts *i)* a textual description and *ii)* an image as inputs, then encodes both into a shared embedding space. The image encoder is a vision transformer that accepts a fixed-size 224x224 image as input, splits it up into 32x32 pixel patches, and then encodes the patches into a 512-dimensional embedding. The text encoder is a transformer that accepts text sequences of up to 77 tokens (each token roughly corresponds to a word) and also produces a 512-dimensional embedding. The outputs of these two encoders are often used to produce a single numerical value, which is computed as the dot product of the image and text embeddings. CLIP's dot product output can be interpreted in many ways, with a common one being the semantic similarity of the two inputs *e.g.,* the text "a dog" and an image of a dog would produce a high score. CLIP was trained on roughly 400 million pairs of images and text captions scraped from the internet, which it used to learn these semantic associations. While CLIP is often successful in general image classification or association tasks, these internet crawls often lack data for more domain-specific tasks such as understanding images taken by satellites, autonomous vehicles, and medical images [260]. As we show in our baseline evaluation, CLIP also performs poorly on UI screenshots, which are relatively rare in the model's original training data.

The purpose of our training procedure is to finetune the *CLIP B/32* to *i)* improve relevance scoring among UI screenshots and descriptions and *ii)* incorporate design quality as a factor in the score, and *iii)* associate descriptions of design defects with screenshots of UIs that contain them. We refer to our model as UIClip, since it is a descendant of CLIP that is optimized for UIs.

9.4.1 Training

We trained UIClip in four stages that incorporated different data sources and training objectives, which were designed for different use cases and tasks. In the first training stage, which we refer to as "pre-training," we trained UIClip using the JitterWeb dataset and the same training objective used in the original CLIP implementation [260]. We found this useful for applications related to retrieval and associating UI screenshots with relevant descriptions. In the second stage, we switched UIClip's training objective to an alternative loss function that specifically focuses on distinguishing good from bad UI designs. These two stages are then repeated for the BetterApp dataset, where each stage uses model weights from the previous stage as a starting point. During all stages of training, we adopt a pre-processing methodology similar to the one used in the original CLIP paper [260] and subsequent reproductions [69], where a random-crop strategy is used to capture different parts of UI screenshots.

CLIP Pretraining Objective

During the pre-training stage, we used the same training objective as the base CLIP model [260], which is described by Equation 9.1.

$$\mathcal{L}_{CLIP} = -\sum_{i} \ln \frac{e^{v_i \cdot w_i}}{\sum_j e^{v_i \cdot w_j}} - \sum_{j} \ln \frac{e^{v_j \cdot w_j}}{\sum_i e^{v_i \cdot w_j}} \tag{9.1}$$

Where, w_i refers to the i-th text embedding in the batch and v_j refers to the j-th image embedding in the batch.

To give a high-level overview of the process, this training objective involves repeatedly sampling a *minibatch* of N examples from the training dataset, where each example consists of an image (UI screenshot) and a textual description (caption with a design quality tag and applied jitters). The model generates embeddings for all text $w_{1...N}$ and images $v_{1...N}$ in the minibatch, then computes an NxN similarity matrix between all combinations of images and text. The objective then computes the cross entropy loss to match each image with its original text description, and vice versa. The intuition behind this process is that the representations of corresponding images and text will gradually become more similar in the shared embedding space, while mismatched pairs will be pushed apart. In the case of UIClip, screenshots will be matched to textual descriptions containing the appropriate design quality tag, design suggestions, and caption.

Pairwise Contrastive Objective

A drawback of the standard CLIP objective is that the minibatches used to compute its loss are randomly sampled from the entire training dataset. Because the size of a minibatch is much smaller than the size of the entire training dataset, there is very low chance that a minibatch will contain examples of closely-related UI screenshots *e.g.,* both a jittered and non-jittered version of a webpage. We hypothesized that this would make it more difficult for the model to learn relationships between these related UIs, which is necessary for assessing the relative quality of related designs. Therefore, we modified the training objective to explicitly compare pairs of related UI screens. Our method is similar to previous methods for pairwise contrastive learning [131], but we use a cross-entropy loss, which is more compatible with the pre-training objective, instead of the margin-based one.

This training objective, shown in Equation 9.2, trains the model so that the embedding of the preferred screenshot has a higher dot product with a text description indicating good design (*i.e.,* a design quality tag of "well-designed") than the embedding of the non-preferred screenshot.

$$\mathcal{L}_{pair} = -\ln \frac{e^{v^{+} \cdot w^{+}}}{e^{v^{+} \cdot w^{+}} + e^{v^{-} \cdot w^{+}}} \tag{9.2}$$

Where v^{+} refers to the embedding of the preferred screenshot, v^{-} is the embedding of the non-preferred screenshot, and w^{+} is the embedding of the text description.

9.4.2 Inference

Preprocessing

CLIP has a fixed image input size of 224x224 pixels, which presents challenges for encoding UI screenshots during inference given that many mobile apps are designed with high height-to-width aspect ratios and dimensions can vary significantly between UIs captured on different devices. Naive pre-processing methods such as image scaling or image cropping can result in significant distortion or exclude important information. One way to address this is to make architectural changes to the model, using similar strategies to previous work [180]. We adopt a simpler strategy to handle variable image sizes using a sliding window strategy. The input screenshot is first resized so that its smaller dimension is equal to 224 pixels. A 224x224 window slides across the larger dimension of size d where the number of evenly-spaced steps is equal to $\lfloor \frac{d}{224} \rfloor + 1$, so that the entire image is covered with the minimal amount of overlapped area. The image encoder is used to compute an embedding for each window of the screenshot, and then all embeddings are averaged together.

UIClip Score

The UIClip score represents a combination of the relevance of the text description and the UI screenshot and the design quality of the UI screenshot. Computing the UIClip score requires *i)* a screenshot of the UI to be evaluated and *ii)* a user-provided caption describing the intended purpose of the UI. A full textual description is constructed by pre-pending a prefix “ui screenshot. well-designed. ” to the user-provided caption. The resulting score between the encoded screenshot and the encoded full description represents a score that describes how well the screenshot adheres to a “well-designed” UI with the target caption.

Design Suggestions

During training, UIClip learns to associate screenshots with with natural language descriptions containing design defects that are potentially contained within them. However, because UIClip doesn’t contain a decoder network, it cannot directly generate text like other auto-regressive transformers [259].

Instead, we develop an alternative approach that uses UIClip to detect possible design defects in an input screenshot, then surfaces them as warnings to the user to fix. For each possible defect, a natural language description is constructed by pre-pending the corresponding prefix to the caption, *e.g.,* “ui screenshot. poor design. bad text sizing. login screen.” We consider all design defects introduced by our jitter function (*e.g.,* “bad text sizing”) and the four CRAP principles (*e.g.,* “bad alignment”). We computed the similarity score between the input image and these text descriptions that corresponded to design defects. To determine the design defects that are surfaced, we dynamically compute a threshold. The threshold is computed as the image’s similarity score with a caption without any defect tags, *e.g.,* “ui screenshot. poor design. login screen.” Design suggestions can also be limited to a smaller number of categories (*e.g.,* only the four CRAP principles) through pre-defined mappings. For example, since the color noise jitter could affect both contrast and repetition, we map “bad color choice” to warnings for these classes.

9.5 Evaluation

The purpose of our evaluation is to quantify multiple aspects of UIClip’s design assessment capabilities and to compare its performance against several state-of-the-art baseline models and ablation conditions. We focus on tasks that correspond to three use-cases: *i)* design quality assessment, *ii)* design suggestion generation, and *iii)* design relevance. In all three tasks, UIClip outperforms baseline models that are often several orders of magnitude larger.

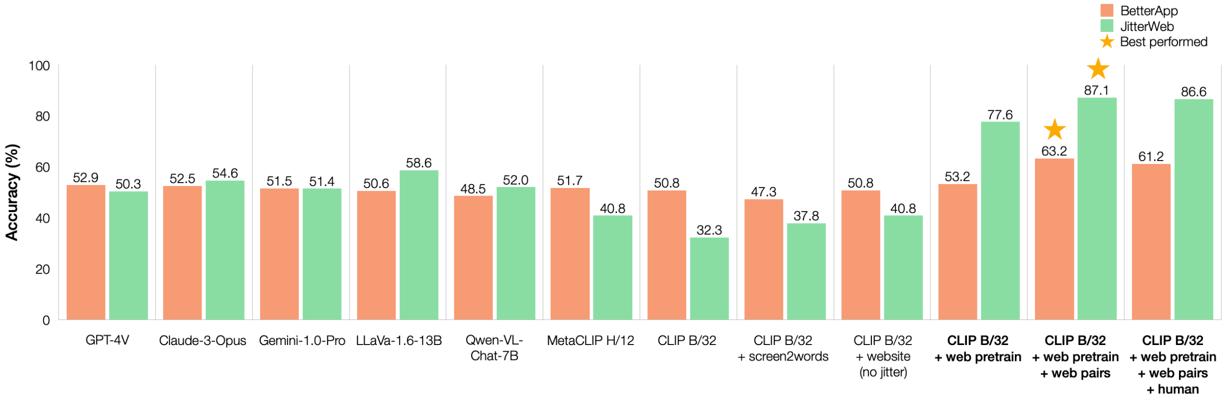


Figure 9.4: Model performance on *design choice* prediction, which involves identifying the preferred UI screenshot from a pair. UIClip models (with bold font) perform the best on held-out human-rated pairs from BETTERAPP and synthetically-generated pairs from JITTERWEB. Most baselines perform poorly, around the level of random chance.

9.5.1 Procedure

We conducted a quantitative evaluation that measured model performance using held-out examples from our datasets.

Baselines

We chose several baselines that consist of different types of multimodal machine-learning models. Originally, we planned to include AIM [246], which is a software package for computing various metrics for UIs. However, there is no definitive way to convert these metrics into design ratings, so we excluded it as a baseline. Therefore, we limit our analysis to the machine-learned models described below.

- **Proprietary Large Vision-Language Models (only accessible via APIs)**
  - *OpenAI GPT-4V* - GPT-4V is a model developed by OpenAI that has been shown to excel at a variety of tasks [23].
  - *Anthropic Claude-3-Opus* - Claude-3-Opus is a model that was introduced by Anthropic at March 2024. It is the largest and most powerful variant among the three Claude-3 models [27].
  - *Google Gemini-1.0-Pro* - Google Gemini-1.0-Pro (Vision) is a model that was introduced by Google in December 2023 [300]. It's the most powerful publicly available model among the three Gemini-1.0 models (Gemini-1.0-Ultra was announced but not publicly available at the time we performed this benchmarking).
- **Open-source Large Vision-Language Models (model weights are publicly available)**
  - *LLaVA-1.6-13B* - LLaVA [212] is a model that was trained using a combination of training examples generated by proprietary large language and vision-language models as well as publicly available visual reasoning and image caption datasets. We used the 13B model (with ViT-L/14 [344] as the vision encoder and Vicuna-13B [70]

as the language decoder) as one of our baselines, which is the largest model that we could fit on our GPU hardware.

  - *Qwen-VL-Chat-7B* - Qwen-VL-Chat [33] is similar to LLaVA, but it was trained using an alternative pre-training objective and datasets. This model (with ViT-bigG [108] as the vision encoder and Qwen-7B [33] as the language decoder) is notable because its pre-training data contained images of documents, which we hypothesized could be relevant for understanding UIs as well.

- **CLIP Models**
  - *CLIP B/32* - We used the unmodified *CLIP B/32* model, which was trained by OpenAI on 400M image-text pairs known as the WebImageText dataset.
  - *MetaCLIP H/12* - Recent research has focused on improving the performance of CLIP models by scaling model size [69] and curating larger and higher-quality multimodal training datasets [108]. *MetaCLIP H/12* is a recent model to achieve state-of-the-art performance for CLIP-like models. It is roughly 6 times larger than *CLIP B/32* and was trained on roughly 6 times more data [336].
- **CLIP Models with Alternative Data**
  - *CLIP B/32 + Screen2Words* - We trained *CLIP B/32* on the largest (to our knowledge) publicly-released dataset of UI screenshots paired with human-authored natural language captions using the default CLIP training objective.
  - *CLIP B/32 + non-jittered websites* - We trained *CLIP B/32* on only non-jittered websites from JITTERWEB using the default CLIP training objective.
- **UIClip**
  - *CLIP B/32 + jittered websites* - We trained *CLIP B/32* on all data from JITTERWEB using the default CLIP training objective.
  - *CLIP B/32 + jittered websites + web pairs* - We trained *CLIP B/32* on all data from JITTERWEB using both the default CLIP objective and the paired contrastive objective.
  - *CLIP B/32 + jittered websites + web pairs + human pairs* - This model consists of *CLIP B/32* trained on JITTERWEB and BETTERAPP using both the default CLIP objective and the paired contrastive objective.

Model Inference

LVLM models rely on a natural-language prompt to instruct them on how to process the image input. We constructed a prompt that asked the model to use the CRAP principles to choose the better design of two UI screenshots and provide the most relevant CRAP principles for its decision. We provided the model with the same short description of the CRAP principles we gave our designers who rated the BETTERAPP dataset. Since some models could only accept one image input, we concatenated two UI screenshots side by side into a single image. In preliminary tests, we verified that all models could distinguish the inputs by asking them to describe the left and right screenshots of the input image.

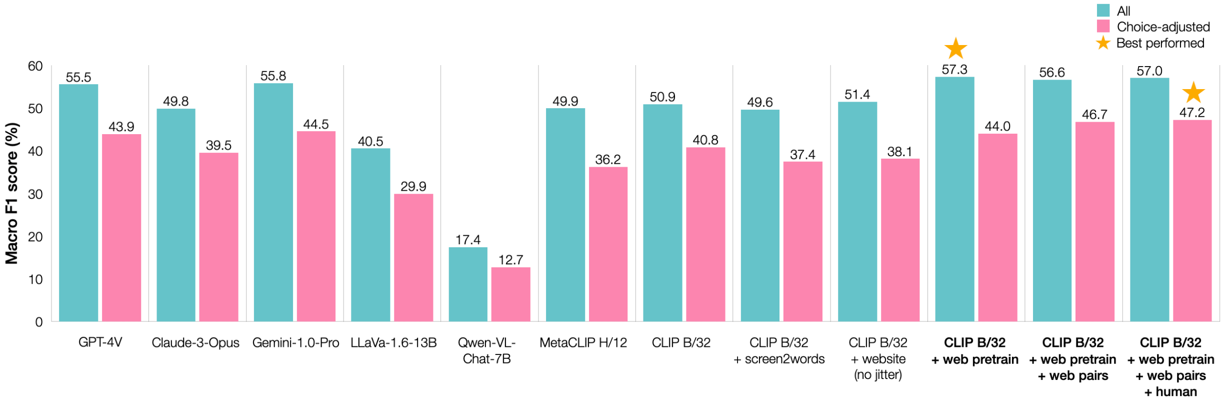


Figure 9.5: Model performance on *design suggestion* prediction, which involves generating design suggestions for a UI based on detected design flaws. We used the macro-averaged F1 score to measure performance across four CRAP principles. In addition, we introduce a choice-adjusted metric that ignores generated suggestions if they led to the incorrect choice. Using both metrics, UIClip models (with bold font) perform the best on held-out pairs from BETTERAPP and JITTERWEB.

We iterated through several versions of prompts which included well-known strategies for eliciting step-by-step reasoning [163]. We chose the best natural language prompt from our tests and used it for all models. The format of the prompt is provided in the appendix (We also included an example of GPT-4V's output when accessed through the web interface in Figure 9.6, with a slightly modified prompt that allowed it to provide additional reasoning). We used the default parameters (e.g., temperature and top-p) for all the LVLMs we compared. UIClip and other CLIP-based models used the inference strategies described in Section 9.4.2.

9.5.2 Results

We focused on evaluating three aspects of design assessment: *i)* UI design quality assessment, *ii)* design suggestion generation, and *iii)* design relevance.

Design Quality

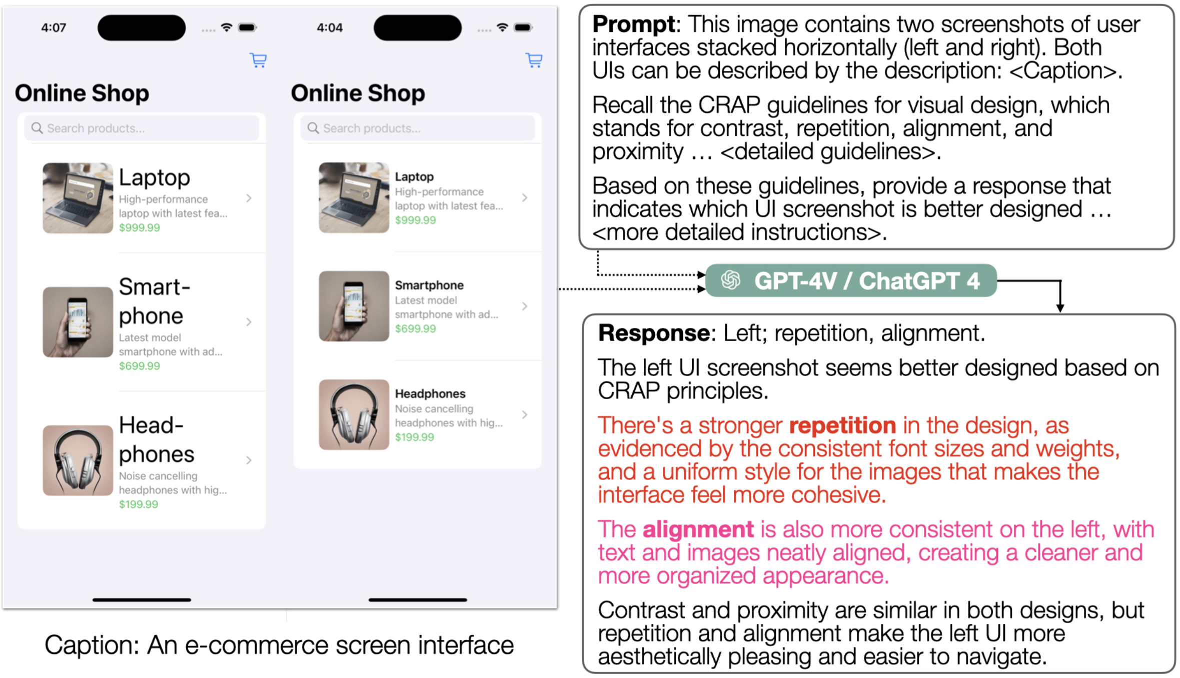
We evaluated a model's *design quality* assessment by measuring its accuracy in identifying the "preferred" UI from an example pair. The results of our experiments are shown in Figure 9.4.

In general, design quality assessment is a difficult task for all tested models, especially when evaluating human-rated pairs of real app screens. Our results are shown in Figure 9.5. For both BETTERAPP and JITTERWEB, the UIClip variant trained with web pairs performed the best, with an average overall accuracy of 75.12%. In particular, we see large improvements in detecting design defects in web pages (87.1%), which is what the majority of the training process and data focused on.

These improvements are notable because *CLIP B/32*, which was the base model of all UIClip variants, performed the worst out of all baselines. *CLIP B/32* performed especially poorly for jittered websites, where a further analysis revealed it erroneously associated certain types of jitters (*e.g.,* dark, unreadable backgrounds) with better design. This suggests that our training procedure and data are effective for learning design quality, especially when compared to other publicly available sources of captioned UIs (*e.g.,* Screen2Words) or general-purpose multi-modal data (*e.g., MetaCLIP H/12*).

Incorporating human ratings appeared to lead to slightly degraded performance (overall accuracy of 73.88%), possibly due to insufficient data. As noted previously, the UIClip with CLIP pre-training objective alone was less effective at improving design quality assessment capabilities because paired UI examples are often not found in randomly sampled minibatches. Nevertheless, it had the third-highest overall accuracy of 65.42%.

Despite their much larger size, all LVLMs performed very poorly on design quality assessment, often around the level of random guessing. Interestingly, GPT-4V (overall 51.58%) refused to provide a response for around 10% of examples, stating "I'm sorry, I can't help you with that." In this regard, open models such as LLaVA performed better than GPT-4V, even though LLaVA was trained by distilling GPT-4V output. However, since LLaVA was not trained to refuse requests [211], it ended up with a higher overall performance for design choice prediction (with 54.59% prediction accuracy).


4:07
Online Shop
Search products...
Laptop
High-performance laptop with latest fea...
$999.99
Smart-phone
Latest model smartphone with ad...
$699.99
Head-phones
Noise cancelling headphones with hig...
$199.99
4:04
Online Shop
Search products...
Laptop
High-performance laptop with latest fea...
$999.99
Smartphone
Latest model smartphone with ad...
$699.99
Headphones
Noise cancelling headphones with hig...
$199.99
GPT-4V / ChatGPT 4

Caption: An e-commerce screen interface

Prompt: This image contains two screenshots of user interfaces stacked horizontally (left and right). Both UIs can be described by the description: <Caption>.

Recall the CRAP guidelines for visual design, which stands for contrast, repetition, alignment, and proximity … <detailed guidelines>.

Based on these guidelines, provide a response that indicates which UI screenshot is better designed … <more detailed instructions>.

Response: Left; repetition, alignment.

The left UI screenshot seems better designed based on CRAP principles.

There's a stronger **repetition** in the design, as evidenced by the consistent font sizes and weights, and a uniform style for the images that makes the interface feel more cohesive.

The **alignment** is also more consistent on the left, with text and images neatly aligned, creating a cleaner and more organized appearance.

Contrast and proximity are similar in both designs, but repetition and alignment make the left UI more aesthetically pleasing and easier to navigate.

Figure 9.6: An example of incorrect design assessment and reasoning from GPT-4V. We asked GPT-4V to compare two screenshots. We fed in the same prompt used in our baseline evaluation, except in this case, we allowed GPT-4V to list its reasoning. The screenshot on the right is the original application screenshot. The screenshot left is a variation of the same interface with a design defect where the text is too large and overflows, affecting readability and coherence. GPT-4V erroneously states the left screenshot is better designed and provides irrelevant and incorrect reasoning.

Design Suggestions

We evaluated all models' design suggestion capabilities by comparing the model-generated output to the CRAP principles selected by designers when rating UI quality in BETTERAPP. There were four possible CRAP principles that could have been chosen for each comparison, which we formulate as a multi-label classification problem with four labels. Since designers were allowed to omit reasoning for comparisons, we ignored comparisons where none of the CRAP principles were selected.

Again, design suggestion was a challenging task for all tested models. Some LVLM baselines listed all four CRAP principles for almost every single example, despite being prompted to only choose the most relevant principles. This appears to be consistent with prior work on using LLMs for heuristic evaluation [91], where similar models often provided a large number of irrelevant design suggestions.

In our case, this phenomena led to artificially high recall for models such as Gemini (87.11% recall) and GPT-4V (84.57% recall). Thus, we introduced a *choice-adjusted* F1 metric that ignored models' design suggestions if it led to choosing the wrong preferred UI, *i.e.,* right reasoning but wrong answer. This lowered all models' recall to more realistic levels, *e.g.,* Gemini's recall was lowered to 49.17% and GPT-4V was lowered to 46.58%. Some open LVLM baselines, such as Qwen-VL-Chat, also had trouble following our prompt and often ignored instructions that asked them to provide reasoning for their answers.

Under both methods of calculation, UIClip variants had the best performance, with the web pre-trained variant performing the best when all examples were considered and the full UIClip variant performing the best when adjusted for choice accuracy.

CLIP variants that were trained on other sources of data did not achieve high performance, since their training data did not include information about present design defects that would help them make suggestions.

Design Relevance

Finally, we also evaluated a model's ability to compute UI relevance, which is useful for assessing designs and for various applications that require example retrieval [53, 145].

To measure UI relevance, we adopted a metric commonly used in information retrieval known as mean reciprocal rank (MRR). An embedding is computed for the preferred screenshot in BETTERAPP and JITTERWEB. For each description in the evaluation set, we appended the prefix "ui screenshot. well designed. " and computed its text embedding. The text embedding is used the calculate similarity scores with all screenshots, which is used to sort them in descending order (*i.e.,* highest similarity first). The rank of the first element with the same description was recorded. Since a lower rank is desirable (indicating higher similarity with the corresponding image), MRR (higher is better) is computed as the average of all *reciprocal* ranks. A higher MRR indicates better retrieval performance. Because there is no straightforward way to generate rankings from LVLMs, we only evaluate models based on the CLIP architecture.

The results of our retrieval evaluation are shown in Table 9.1. The variant of UIClip pre-trained on JITTERWEB using the default CLIP objective achieves the highest MRR score for both BETTERAPP (0.3851) and JITTERWEB (0.4085). UIClip variants trained using pairwise

Table 9.1: Model Performance on UI Retrieval for both BetterApp and JitterWeb datasets. The variant of UIClip trained with the CLIP pretraining objective (*i.e.,* without paired data) performed the best, while other variants of UIClip had poor performance, due to the pairwise objective's high prioritization of design choice accuracy.

Model	MRR (BetterApp)	MRR (JitterWeb)
MetaCLIP H/12	0.2722	0.2350
CLIP B/32	0.2534	0.1466
CLIP B/32 + Screen2Words	0.2938	0.1130
CLIP B/32 + Web	0.3467	0.3766
CLIP B/32 + Jit. Web	**0.3851**	**0.4085**
CLIP B/32 + Jit. Web + Web Pairs	0.0962	0.0924
CLIP B/32 + Jit. Web + Web Pairs + Human	0.1096	0.1214

loss were the *worst* performers, with MRRs lower than the original *CLIP B/32* base model, because the objective focuses on the design-comparison task. In our discussion, we provide more detailed reasoning for this phenomenon.

Nevertheless, our evaluation still shows that our datasets are useful for learning design relevance, especially when training objectives are closely aligned. For example, applying the CLIP objective to JitterWeb is much more effective than alternate data sources and nearly doubles $(0.2000 \rightarrow 0.3968)$ the overall retrieval performance of the base *CLIP B/32* model.

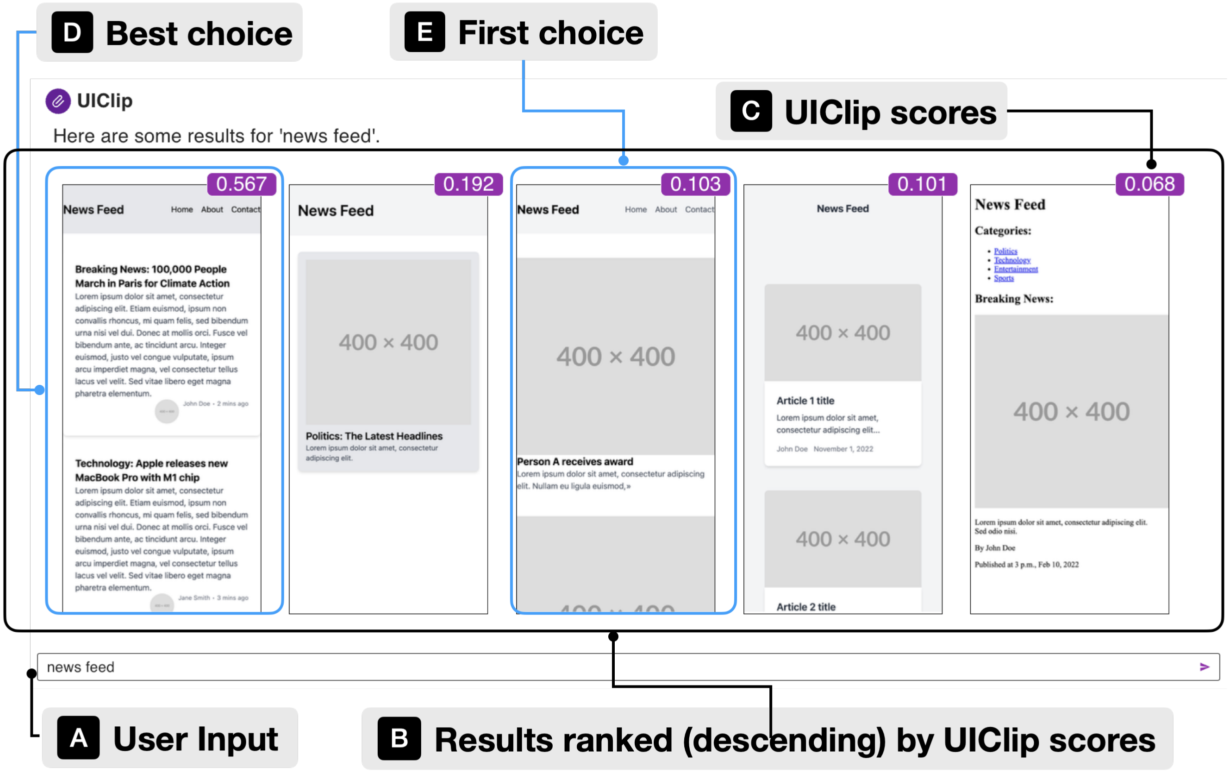
9.6 Example Applications

Based on the three capabilities of UIClip that we evaluated, we present a suite of example applications that illustrate how common user-facing design tools can be enhanced with our model.

9.6.1 Improving UI Code Generation

We built a web application that allows users to generate rendered UI screenshots from a natural language description of a UI. To use the interface (Figure 9.7), users enter their descriptions in a textbox, which is formulated in a prompt. The prompt is fed into an external LLM (OpenAI GPT-3.5), which generates web code (HTML/CSS) using the description. We sampled $n = 5$ different outputs and rendered each into a screenshot using the script that programmatically controlled a browser. If the web code referenced external images, we replaced them with a placeholder image to render. Screenshots were fed into UIClip and were scored against the input prompt. The screenshot of the results ranked in descending score order is returned to the user.

This is a simple example of how UIClip could be used to improve the output of generative models, most similar to existing "best-of-n sampling" approaches. The method can also be incorporated into the additional vision checkup [282] and used for feedback in self-improving generative model outputs [220]. Our technique is simple to implement and does not require access to the underlying model's weights; however, it is computationally expensive during infer-


UIClip

Here are some results for 'news feed'.

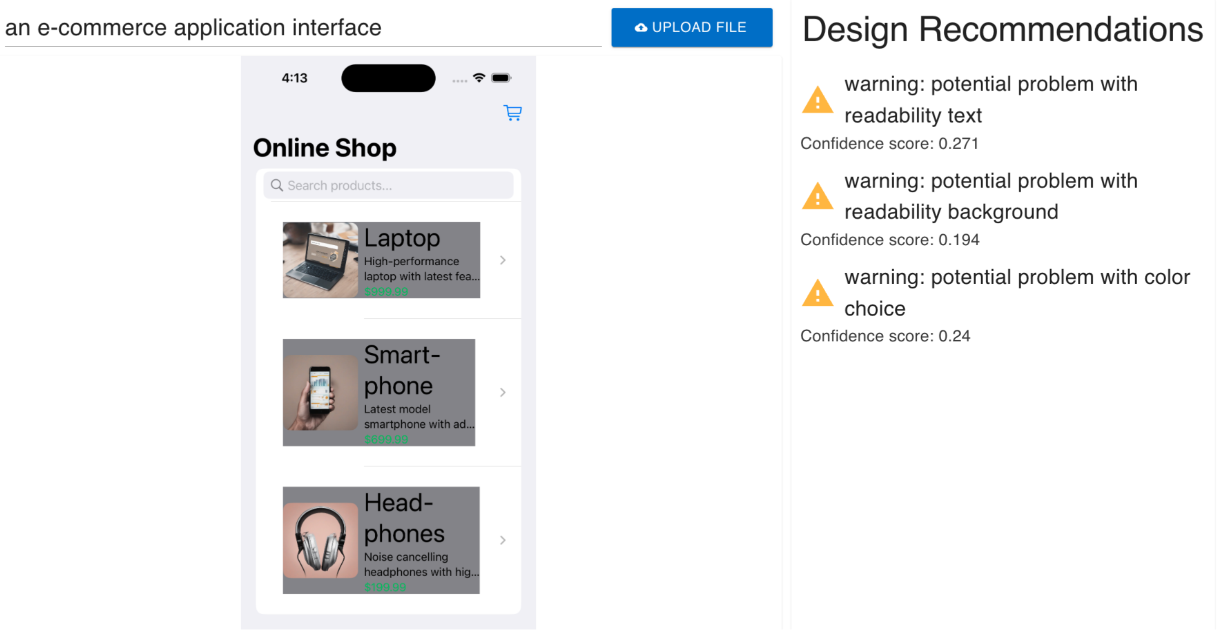
news feed

Figure 9.7: UI Generation Example Application. UIClip is used to rank rendered UI code provided by an external LLM (GPT-3.5). The user can describe a UI (A) and compares different LLM-generated results (B). We ranked these options based on UIClip's scores and displayed them alongside the rendered screenshot (C). Note that UIClip ranked the first screenshot on the left as the highest quality (D), while we received the third screenshot from the left as the first choice output from our LLM (E).

ence because multiple candidate solutions must be generated. If model weights are available, this process could be further improved by applying UIClip's filtering during the training process of generative models, or if UIClip was used as a reward model in reinforcement learning fine-tuning approaches [127, 250]. We leave these investigations to future work.

9.6.2 UI Design Tips

We built a tool that allows users to upload screenshots of UI designs to generate design tips using our model's design suggestion capabilities. We developed a web application (Figure 9.8) that allows users to upload a screenshot of an app or UI design, and our system generates tips that are surfaced to the user, although a similar idea could be better integrated into, for example, UI design applications for improved ease-of-use. One limitation of our current application is that it might suggest improving text contrast, but it is unable to provide additional information for which part of the UI led to the recommendation. This is due to our problem formulation that pairs descriptions with entire screenshots and doesn't contain location information. Future


an e-commerce application interface
UPLOAD FILE
4:13
Online Shop
Search products...
Laptop
High-performance laptop with latest fea...
Smart-phone
Latest model smartphone with ad...
Head-phones
Noise cancelling headphones with hig...
Design Recommendations

warning: potential problem with readability text
Confidence score: 0.271

warning: potential problem with readability background
Confidence score: 0.194

warning: potential problem with color choice
Confidence score: 0.24

Figure 9.8: UI Design Tips Example Application. We use our design suggestion generation algorithm to generate design suggestions from a user-provided description and user-uploaded UI screenshot. This example shows suggestions to improve the readability of text and color choice for the application.

improvements can help address this by collecting the relevant data and incorporating that into text descriptions, or by sliding a smaller window across the UI screenshot and associating generated design suggestions to the location of the window. We leave these additional features to future work.

9.6.3 UI Example Retrieval

UI design search has been explored by many prior works [53, 80, 170], and it has the potential to accelerate the design process by providing relevant examples that designers use to seek inspiration during early phases of the design process. Existing systems built for UI example retrieval have focused on querying and indexing UI screenshots by their layout (*e.g.,* to support finding designs similar to an exemplar) or captions (*e.g.,* to support natural-language or free-form search). Since UIClip contains both an image and text encoder, it is possible to support both of these use cases, although our example application focuses on handling text-based queries. Our application uses a similar procedure to our UI relevance evaluation, where model-computed embeddings are used to retrieve and sort screenshots based on the user's query. UIClip's score can take into account both the relevance and quality of retrieved examples, and we incorporate a *negative prompt* that biases the query vector away from simple or ambiguous designs [272].

We built a web application that contains a search box where the user enters their query. Figure 9.9 shows examples of screens retrieved for a set of queries indexed by UIClip and the vanilla CLIP model.

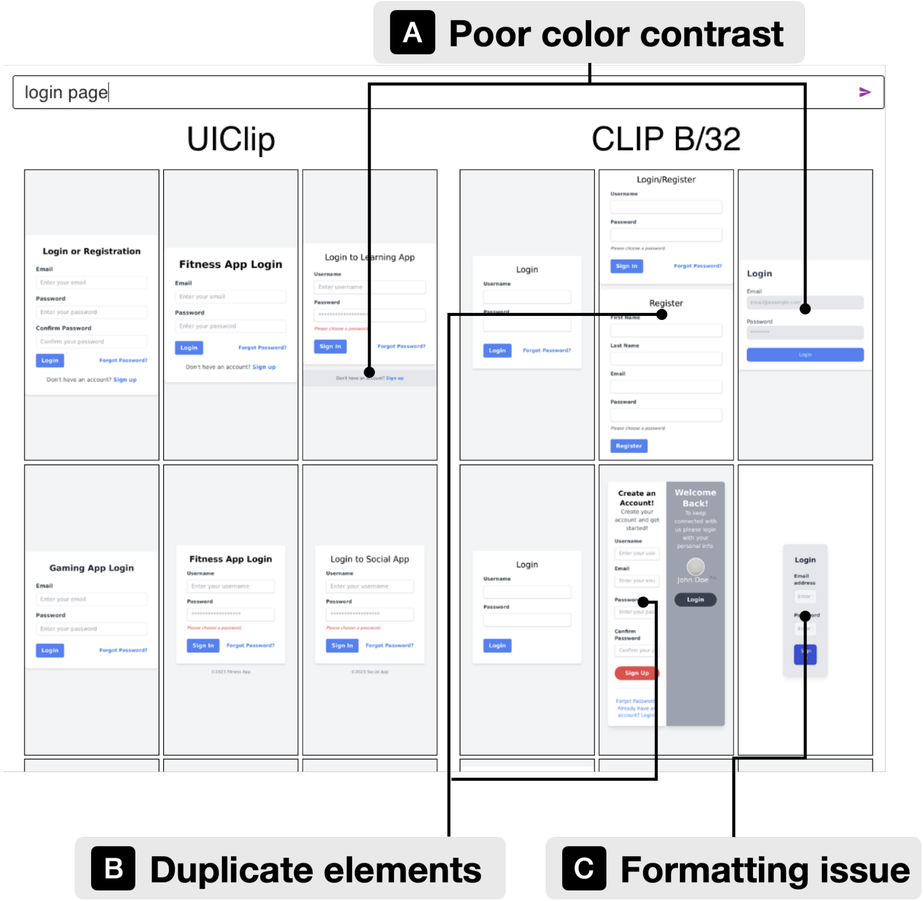


Figure 9.9: UI Example Retrieval Example Application. In this use case, the designer searches for examples of login screens queried from a set of LLM-generated UIs, many of which have design flaws. While both UIClip and *CLIP B/32* gives a diverse range of applications, we see that there are more design flaws present from *CLIP B/32*. Some screens exhibit poor color contrast (A) which may imply that the component is disabled, duplicate elements (B) that can confuse end users, and overall poor formatting and overflow layouts (C).

9.7 Discussion

Our experiments and example applications show that UIClip's design assessment capabilities can improve many machine-assisted design tools. In this section, we discuss UIClip's implications, limitations, and directions for future work.

9.7.1 Data-driven Learning of UI Design

Our paper introduces techniques for machine-learning a *generalized* scoring function (*c.f.* personalized functions [111]) that quantifies aspects UI design. We discuss our work's data and algorithm contributions.

We hypothesized that a large volume of data (millions of examples) is important for effectively learning to assess designs, similar to how seasoned human designers develop their intuition through years of experience. This hypothesis was largely supported by our experimental results. We showed the substantial benefits of training on our large-scale dataset of UI designs, leading to improvements over alternate datasets (*e.g.,* Screen2Words [315]) that more *human-authored* descriptions but fewer overall samples. When we incorporated our own human-rated BETTERAPP dataset, we found that in most cases, it did not result in substantial changes, most likely due to insufficient data volume.

At the same time, dataset size alone is not enough to ensure good design assessment performance. For example, *MetaCLIP H/12* was trained on a total of 2.5 billion pairs. Our paper introduces training objectives targeted at different aspects of design quality. We employed two objectives for training UIClip, a batch-wise contrastive objective (*i.e.,* CLIP's pretraining objective) and a pairwise contrastive objective, designed specifically for quality comparisons. Based on our results, the pairwise objective represents a tradeoff where it achieves higher focus on design-comparison tasks by focusing on pairs of relevant screens but incurs a penalty on retrieval-related tasks, since it is not trained to distinguish irrelevant examples from a diverse minibatch. We leave further investigation of how to optimally combine these two training objectives to future work; although given the relatively small size of our model, we believe it would be feasible to use different variations for application-specific scenarios.

9.7.2 Formulating UI Design Quality

UIClip's current model architecture is designed around the assumption that design quality can be represented by a numerical score. However, there are many nuances that cannot be captured by this formulation.

Within the context of our collected data, we distinguish between assessing UIs for *design defects* and understanding more subtle *design preferences.* JITTERWEB was constructed by introducing "jitters" into web pages, that intentionally violate design guidelines. In these cases, one might expect to more objectively identify the preferred screen, since the alternative screen would contain a defect. We found this case well captured by our formulation, as shown by our models' higher performance on the JITTERWEB test data. In contrast, examples from BETTERAPP are more representative of *design preferences.* Many of its samples were real-world apps, which are often designed professionally and less likely to contain design defects. To verify this, we analyzed design quality performance on the subset of synthetic, LLM-generated screens within BETTERAPP and compared it with the app screens from VINS. Many of the LLM-generated screens (as shown in Figure 9.9) contain design defects, which potentially led to easier comparisons. UIClip's accuracy for rating the quality of synthetic screens (67.65%) was much higher than for real app screens (57.89%). This trend was true for almost all other tested models, where the average of all models' accuracy on synthetic screens (56.05%) was higher than real apps (51.50%). It is also possible that UIClip is not trained to detect certain types of design defects present in real-world apps, *e.g.,* violations that require the semantic understanding of content, such as information flow hierarchy.

Besides the nature of design defects in real-world apps, design preferences could also vary by person. For example, it is reasonable to expect that someone who frequently uses iOS devices might feel more familiar with iOS screenshots over Android ones. The design language of the same platform can change over time, causing corresponding shifts in user perception, *e.g.,* some screenshots in VINS were from the older Rico dataset [80]. While UIClip is currently not designed to support more personalized use cases, we envision that it could be finetuned with user-provided preference paired [111] or augmented so that it incorporates platform-specific design into its prompt, *e.g.,* "android material design screenshot. well-designed."

9.7.3 Supporting UI Design Applications

Related to our problem formulation is the types of design-assistance tasks that UIClip can support. Despite only producing a numerical score as output, we introduced inference techniques that extend beyond simple UI scoring and allow for a limited generation of natural language design suggestions. We developed three example applications that demonstrate how UIClip could facilitate some forms of automated design assistance. While we did not conduct formal user evaluations of our example applications, we built these applications based on validated systems described in the literature, which suggest they would provide value to users.

However, there are many types of design assistance that are not yet supported by UIClip. For example, while UIClip can infer the presence of design defects in a screenshot, there is currently no straightforward method to localize them (*e.g.,* footer bar has poor color contrast). We believe this capability is important for practical use since it provides cues for designers to address the detected flaws. One promising approach, previously applied to other visual design tasks [277], is to augment our current model with model explainability frameworks to understand which parts of the image contribute to predictions. Future iterations of the UIClip could also be trained on sub-windows of a UI for finer-grain inference of fault location, similar to how object detection architectures work. Finally, UIClip could be fine-tuned with more detailed natural language descriptions that associate spatial information with predicted flaws (*e.g.,* "bad color contrast in footer bar") or even provide suggested fixes (*e.g.,* "bad color contrast. make footer darker"), although this would necessitate a more complex inference algorithm.

Recent trends in machine learning suggest that model architectures that generate free-form text can be more easily scaled and provide more flexible feedback. Our evaluation found that VLMs generally performed poorly, and prior work suggests that LLMs are prone to providing irrelevant design suggestions [91]. A qualitative assessment of current LLM responses (Figure 9.6) suggests that current LLMs produce realistic-sounding but inaccurate reasoning. However, we believe that our work could be useful in improving foundation models such as LVLMs. For example, the UIClip model could be used as a "reward model" that guides their UI generation (Section 9.6.1) and design assessment capabilities. Furthermore, our datasets could be reformatted and used to fine-tune LVLMs for UI-related tasks [184].

Finally, most machine learning models, including UIClip and LVLMs are limited in that they can only process a single state of the UI (*i.e.,* a screenshot) when responding to text prompts. Because of this limitation, our current approach focuses on assessing the visual design of a single screen using the CRAP visual design principles. A more holistic evaluation of both UI design and usability depends on a deeper understanding of interface functionality and app navigation flows, which requires both observation and interaction. To support this, we envision that models like UIClip, could be integrated into interactive systems, such as crawlers, that can interact with and explore different parts of an entire app [298, 328].

Table 9.2: Hyperparameters of models and algorithms used in our paper.

Algorithm	Hyperparam.	Value
UIClip (JitterWeb pretraining)	Batch Size	128
	Epochs	1
	Learning Rate	$5e-7$
	Weight Decay	0.2
	Adam Beta 1	0.9
	Adam Beta 2	0.98
	Adam Epsilon	$1e-6$
UIClip (JitterWeb pairwise, BetterApp)	Batch Size	256
	Epochs	1
	Learning Rate	$5e-7$
	Weight Decay	0.2
	Adam Beta 1	0.9
	Adam Beta 2	0.98
	Adam Epsilon	$1e-6$
BetterApp DBSCAN	Epsilon	0.1
	Min samples	5
	Metric	Cosine Similarity

9.8 Conclusion

In this paper, we introduce a computational model called UIClip that automatically assesses various aspects of UI design: *i)* design quality, *ii)* UI relevance, and *iii)* is capable of generating design suggestions based on predicted defects. Our model is trained from a large-scale dataset of 2.3 million UIs that we collected and augmented with synthetic and human ratings of design quality. In an evaluation with several strong baselines, we demonstrate our model's performance in UI design understanding in our three target UI tasks, showing that UIClip outperforms all other baselines in all tasks. Finally, we introduce three example applications that demonstrate how UIClip can facilitate novel applications through its automated design assessment capabilities: *i)* UI code generation, *ii)* UI design tips generation, and *iii)* quality-aware UI example search. Overall, our work demonstrates the process of effectively encoding design awareness into the computational models.

9.9 Hyperparameters

Table 9.2 provides hyperparameters for various models and algorithms used in our paper. Our CLIP training hyperparameters were based on values from the original CLIP paper [260], and were manually adjusted to fit on our hardware and based on performance observations.

9.10 Large Vision-Language Model Prompt

Below, we provide the prompt that was used to evaluate UI screenshots in our quantitative study. The prompt below is used to evaluate a screenshot of an e-commerce application and contains the same description of CRAP guidelines that we gave to human raters.

```
This image contains two screenshots of user interfaces stacked
horizontally (left and right). Both UIs can be described by the
description:

An e-commerce application interface

Recall the CRAP guidelines for visual design, which stands for
contrast, repetition, alignment, and proximity.

The C.R.A.P principles, coined by Robin Patricia Williams in her non-
designers’ design book, are a set of guidelines aimed at improving the
 visual appeal and effectiveness of graphic designs. These principles
are essential for creating visually appealing and user-friendly
designs. CRAP stands for:
Contrast: This principle suggests that elements that are not the same
should be very different so that they stand out. Using contrast can
attract the viewer’s attention and help organize information. It can
be applied through variations in color, size, typeface, and other
visual elements.
Repetition: Repetition involves repeating some aspect of the design
throughout the entire piece. This can include the consistent use of
colors, fonts, and logos, which helps to create a cohesive and
harmonious look. Repetition strengthens a design by tying together
individual elements and can enhance the overall sense of unity.
Alignment: Every element should have a visual connection with
something else on the page. This doesn’t mean that elements always
need to be in a straight line, but rather that they should be visually
 connected in a way that makes the entire design appear well organized
. Proper alignment eliminates disorder, connects elements, and creates
 a visually logical structure.
Proximity: Items that relate to each other should be grouped together,
 which helps in organizing information and reducing clutter. By
effectively grouping related elements, the design becomes easier to
comprehend, and relationships between elements become clearer to the
viewer. Proximity can also help in creating focal points in a design.

Based on these guidelines, provide a response that indicates which UI
screenshot is better designed. The first part of your response should
contain one of two choices: ’left’, ’right.’ The second part of your
response should contain a comma-separated list of which CRAP
principles (if any) are most relevant to your choice. Do not provide
```

explanations, and separate the first and second part of your response with a new line.

10 Conclusions & Future Work

This dissertation investigates a new approach for addressing a fundamental challenge in human-computer interaction: creating interfaces that make computers useful to all people in all contexts. Traditionally, this has been difficult because most UIs have been manually created to be used in pre-defined ways, without much opportunity for adaptation. The assumptions baked into "one-size-fits-all" UIs are often unable to accommodate the diversity of user abilities, usage contexts, and use-cases. My research has shown that by building systems that can computationally understand UIs and how they are used, we can be dynamically optimized UIs to better serve digital content and functionality. I've focused on establishing the technical foundation for this approach, by developing systems for machine-driven *user understanding*, *UI understanding*, and *UI generation*.

10.1 Contributions

To summarize, I've made contributions in each of these three areas. In the area of *user understanding*, I first showed (in Chapter 3) that supporting interface customization alone is not enough to improve user experience, since users often do not know about configurations that they would benefit from, especially accessibility-related features. I show that systems that understand usage context and proactively recommend adjustments have greater potential to improve user experience. In a user study with 19 older adults who used our recommender system while performing everyday tasks, we showed that the majority (74%) of predicted recommendations were rated as helpful. Nevertheless, this approach of mapping usage behaviors to existing features has traditionally been limited, since applications often do not properly expose their semantics to external services, such as accessibility features. In the area of *UI understanding*, I showed that it is possible to overcome this barrier using data-driven ML models that predict interface layout (Chapter 4), structure (Chapter 5), and functionality (Chapter 6) from visual information (*i.e.,* screenshots), which is how UIs are assumed to be used. Finally, I combined both user and UI understanding to *dynamically generate UIs* that meet the specific needs of users. Chapter 7 presents a system that optimizes an existing UI layouts for a user's personalized interaction profile, resulting in 9% faster target acquisition times. Because Chapter 7 focused on optimizing an existing layout, only small changes could be applied without violating pre-existing design constraints, such as alignment and spacing. To this end, I presented techniques for UI code generation (Chapter 8) and automated ML-driven design assessment (Chapter 9) that can result in more significant modifications while maintaining design quality.

10.2 Future Work

My future work will focus on several areas that stem from my initial research.

10.2.1 Computational UI Understanding

Within the framework presented in this dissertation, there are many promising directions for future work.

User Understanding

Understanding users and their context at a higher resolution can facilitate more targeted personalization and adaptation of interfaces. Most work on user understanding, including the work I presented, categorizes human behavior into a fixed number of categories, which limits how this information can used. The system in introduced in Chapter 3 uses a set of heuristics to detect a fixed number of user behaviors, which suggest different accessibility needs. While this is sufficient for simple feature recommendation (*e.g.,* a link that navigates to the font size settings page), it is more challenging to, for example, compute more specific values needed by those features (*e.g.,* the optimal font size). In Chapter 7, I used a higher resolution representation to model the relationship between a user's interaction difficulty (*e.g.,* target acquisition speed and error rate) and a UI's spatial layout, which allowed for more precise adjustments to be applied. However, this work focused on a narrow part of user interaction that could be greatly expanded to enable more complex optimizations.

UI Understanding

Advances in ML techniques, such as large multi-modal models have great shown potential in facilitating more accurate machine understanding of visual inputs, such as UIs. Recent work suggests that large "foundation" models can learn from much larger datasets and are able to detect more complex patterns that arise in the data, allowing them to replace and surpass smaller, domain-specific models that were previously required. A number of challenges remain for incorporating these approaches into UI understanding systems. Although foundation models are often trained on billions of documents and images, only a small fraction of the data consists of UIs, which limits their applicability. In Chapters 4 and 6, I introduced methods for efficiently scaling up the size of UI datasets through automated crawling techniques. Another challenge is assigning semantic labels, such as widget locations and functionality, to these automatically collected examples. Current training pipelines for large language and multi-modal models have focused on collecting labels from specially trained annotators or incorporating user feedback (*e.g.,* thumbs up and thumbs down). In Chapter 6, I explored an approach inspired by "human-like" learning, which allows a model-driven agent to learn from the results of self-guided explorations instead of large pre-labelled datasets. In addition to labeling attributes of static screenshots, these exploration strategies may also allow systems to understand higher-level interactions and flows within applications, rather than single UI states. Similar strategies could also be applied to train large foundation models and improve their understanding of UI data. Thus, a promising future direction for UI understanding is building systems that not only *use* UIs as humans do, but to *learn to use* UIs as humans do.

UI Generation

Generating high-quality UIs is a challenging task that requires a combination of user and UI understanding, and advances in those areas will also improve UI generation. A specific challenge is encoding HCI and UI design concepts into objectives and constraints needed for automated generation. In Chapter 7, I showed that using smaller personalized models trained on user calibration data in conjunction with optimization techniques could lead to small levels of improvements for narrow tasks (*e.g.,* target acquisition). Larger state-of-the-art ML approaches are promising in that they can potentially learn more generalizable evaluation metrics from richer sources of data. However, in Chapter 8 and 9, I showed that foundation models struggle to identify simple design defects (*e.g.,* UIs with poor color contrast) and fail to learn the relationship between descriptions of UIs and their visual representations In both cases, I proposed synthetic data generation and training techniques to improve existing foundation models, and a promising future direction is to continue to improve the UI evaluation capabilities of these models.

10.2.2 Expanding Opportunities for Interface Improvement

The framework I present in this dissertation is targeted at applying computational UI understanding to improve user experience with existing applications. I've shown that ML models can often successfully understand and operate UIs, separating an application's content and functionality away from its "default" interaction. However, in many cases, it unclear what the best replacement interaction or interface is. Within the bounds of traditional 2-D graphical UIs, this could mean shifting the placement of a UI layout or mapping data and actions to alternative widgets. Thus far, my work has explored these possibilities using optimization techniques (Chapter 7) and code generation (Chapter 8). A promising future direction is to further investigate this space of interface improvements and associated tradeoffs. Finally, as emerging computing modalities such as mixed reality and conversational agents become more prevalent, they could provide new flexibility in presenting applications. Given that these are still areas of active research and development, this are opportunities for defining interactions and conventions that are accessible and context-aware from the ground up.

10.2.3 UI Design and Development Tools

Another area of future work is building tools that makes it easy for designers and developers build higher quality interfaces that adhere to best practices. The models I've developed for detecting and repairing UIs during application runtime can be used across in the development process, aiding UI design, implementation, and testing. Similar to assistive technology, much of the existing development infrastructure and tooling (*e.g.,* UI tests and app monkeys) can be made more robust by reducing reliance on metadata provided by specific toolkits. This could also allow third parties (*e.g.,* mobile app stores) to crawl submitted apps and verify that meet usability and accessibility criteria. Since many of the systems I developed are designed to perceive UIs using the same modalities as most humans do (*i.e.,* through visual information), they could serve as useful predictors of future user experience. For example, model predictions could be used to verify human-authored designs, *e.g.,* a button does not visually appear to be tappable, which could suggest that stronger visual signifiers are needed. Chapter 9 presents a ML-model to assess the aesthetic quality of UI designs, which was trained on, among other things, ratings from 12 human designers. Both these examples are aimed at providing rapid and actionable feedback at early stages of design, where it is normally difficult to obtain feedback. Finally, my work on UI program generation can reduce the friction UI mockup design and implementation (*i.e.,* design handoff), facilitating more rapid prototype iteration and testing. For all these avenues, further research is needed to understand how best to design these tools so that they fit the developer preferences and workflows.

10.2.4 Bridging HCI and Computational Techniques

Finally, my work has demonstrated a broader need for formulating HCI concepts (*e.g.,* usability, accessibility, and visual aesthetics) in ways that are compatible with state-of-the-art computational approaches (*e.g.,* machine learning, optimization). This is challenging because in HCI, they have traditionally been measured using qualitative and subjective methodologies (*e.g.,* user studies), while most computational approaches requires them to be deterministic and quantifiable, *e.g.,* objective functions. Chapters 7 and 9 in this dissertation show my initial work towards this goal, which has involved using data collected using HCI methodologies (*e.g.,* a user study) to train machine learning that can rapidly simulate human feedback. Both examples showed promising results but also highlighted several challenges, such as matching the expressiveness of human feedback and preserves the composition of individual feedback (instead of "regressing to the mean"). Another challenge stems from the data requirements of recent "large" models with millions or billions of parameters. To this end, I've released several large-scale datasets (Chapters 4 and 9) from my work and developed approaches (Chapters 4 and 6) for sustainably and efficiently collecting updated examples of UIs, which often undergo content and design refreshes. In future work, I plan to continue research in bridging methodologies and techniques between these two fields.

Bibliography

[1] CforAT accessibility consultant. https://www.cforat.org/about/at_specialist. Accessed: 2021-01-17. 2.1.2

[2] Google digital wellbeing tools. https://wellbeing.google/tools/. Accessed: 2021-01-17. 3.7

[3] Apple Developer arfacetrackingconfiguration. https://developer.apple.com/documentation/arkit/arfacetrackingconfiguration, . Accessed: 2021-01-17. 3.5.1

[4] Apple Support use notifications on your iphone, ipad, and ipod touch. https://support.apple.com/en-us/HT201925, . Accessed: 2021-01-17. 3.7

[5] Apple Newsroom watchos 6 advances health and fitness capabilities for apple watch. https://nr.apple.com/d2i8t9c2n9, . Accessed: 2021-01-17. 3.5.2

[6] Spaulding Rehab assistive technology. https://spauldingrehab.org/conditions-services/assistive-technology. Accessed: 2021-01-17. 2.1.2

[7] Craig Hospital assistive technology. https://craighospital.org/services/assistive-technology. Accessed: 2021-01-17. 2.1.2

[8] Pollfish real consumer insights. https://www.pollfish.com/. Accessed: 2021-01-17. 3.3

[9] A profile of older americans - 2017. https://acl.gov/sites/default/files/Aging%20and%20Disability%20in%20America/2017OlderAmericansProfile.pdf. Accessed: 2021-01-17. 3.2, 3.3

[10] Web Axe detecting screen readers - no. https://www.webaxe.org/detecting-screen-readers-no/. Accessed: 2021-01-17. 3.2

[11] Apple Support about siri suggestions on iphone. https://support.apple.com/guide/iphone/about-siri-suggestions-iph6f94af287/ios. Accessed: 2021-01-17. 3.7

[12] Apple Developer Documentation uiaccessibility. https://developer.apple.com/documentation/uikit/uiaccessibility. Accessed: 2021-01-17. 3.5.4

[13] Who study on global ageing and adult health (sage). https://www.who.int/healthinfo/sage/en/. Accessed: 2021-01-17. 3.6.2

[14] Apple Support zoom in on the iphone screen. https://support.apple.com/

guide/iphone/zoom-in-on-the-screen-iph3e2e367e/ios, . Accessed: 2021-01-17. 3.4

[15] Apple Support about the accessibility shortcut for iphone, ipad, and ipod touch. https://support.apple.com/en-us/HT204390, . Accessed: 2021-01-17. 3.5.3

[16] Apple Support about the buttons and switches on your iphone, ipad, or ipod touch. https://support.apple.com/en-us/HT203017, . Accessed: 2021-01-17. 3.5.3

[17] Using your mac classic. Accessed: 2021-01-17. 3.2

[18] What is the ideal screen size for responsive design? https://www.browserstack.com/guide/ideal-screen-sizes-for-responsive-design, 2022. Accessed: 2022-09-15. 4.3.1

[19] Chrome DevTools engineering blog full accessibility tree in chrome devtools. https://developer.chrome.com/blog/full-accessibility-tree/, 2022. Accessed: 2022-09-15. 4.3.1, 4.4.1

[20] Puppeteer - chrome. https://developer.chrome.com/docs/puppeteer/, 2022. Accessed: 2022-09-15. 4.3.1

[21] Bootstrap, 2023. URL https://getbootstrap.com/. Accessed 2023-08-19. 2.1.1

[22] Webflow, 2023. URL https://webflow.com/. Accessed 2023-08-19. 2.1.1

[23] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. 9.5.1

[24] Adobe. Adobe dreamweaver, 2023. URL https://www.adobe.com/products/dreamweaver.html. Accessed 2023-08-19. 2.1.1

[25] Airbnb. Sketching interfaces, 2017. URL https://airbnb.design/sketching-interfaces/. 2.2.2

[26] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Merouane Debbah, Etienne Goffinet, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. Falcon-40B: an open large language model with state-of-the-art performance. 2023. 8.3.1

[27] Anthropic. Introducing the next generation of claude. https://www.anthropic.com/news/claude-3-family, 2023. Accessed: 2024-04-01. 9.5.1

[28] Apple. Testing for accessibility on os x, 2015. URL https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html. 8.6.1

[29] Deniz Arsan, Ali Zaidi, Aravind Sagar, and Ranjitha Kumar. App-based task shortcuts for virtual assistants. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 1089–1099, 2021. 2.2.2

[30] AutoIt. Function pixelsearch, 2021. URL https://www.autoitscript.com/autoit3/docs/functions/PixelSearch.htm. 2.2.2

[31] Yuliya Bababekova, Mark Rosenfield, Jennifer E Hue, and Rae R Huang. Font size and viewing distance of handheld smart phones. *Optometry and Vision Science*, 88(7):795–797, 2011. 3.5.1

[32] Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, and Blaise Agüera y Arcas. Uibert: Learning generic multimodal representations for ui understanding. In *International Joint Conference on Artificial Intelligence*, 2021. 2.2.2

[33] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond. 2023. 9.4, 9.5.1

[34] Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A Smith. Training with exploration improves a greedy stack-lstm parser. *arXiv preprint arXiv:1603.03793*, 2016. 5.5.2

[35] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005. 2.2.3

[36] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 211–221. IEEE, 2016. 7.6.1

[37] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. In *ACM Transactions on Graphics (ToG)*, volume 28, page 24. ACM, 2009. 7.4.4

[38] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 1–6, 2018. 2.2.2, 2.2.3

[39] Joanna Bergstrom-Lehtovirta and Antti Oulasvirta. Modeling the functional area of the thumb on mobile touchscreen surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1991–2000, 2014. 2.2.1, 7.4.1

[40] Carlos Bernal-Cárdenas, Nathan Cooper, Madeleine Havranek, Kevin Moran, Oscar Chaparro, Denys Poshyvanyk, and Andrian Marcus. Translating video recordings of complex mobile app ui gestures into replayable scenarios. *IEEE Transactions on Software Engineering*, 2022. 4.4.3

[41] Marcelo Bertalmio, Andrea L Bertozzi, and Guillermo Sapiro. Navier-stokes, fluid dynamics, and image and video inpainting. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001. 7.4.4

[42] Xiaojun Bi, Yang Li, and Shumin Zhai. Ffitts law: modeling finger touch with fitts' law. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages

1363–1372, 2013. 2.2.1

[43] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR, 2023. 9.3.1

[44] Pavol Bielik, Marc Fischer, and Martin Vechev. Robust relational layout synthesis from examples for android. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA): 1–29, 2018. 7.7.3

[45] Jeffrey P. Bigham. Making the web easier to see with opportunistic accessibility improvement. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 117–122, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3069-5. doi: 10.1145/2642918.2647357. URL `http://doi.acm.org/10.1145/2642918.2647357`. 3.2, 3.5.1

[46] Jeffrey P. Bigham. Making the web easier to see with opportunistic accessibility improvement. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, page 117–122, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330695. doi: 10.1145/2642918.2647357. URL `https://doi.org/10.1145/2642918.2647357`. 7.2

[47] Jeffrey P Bigham, Tessa Lau, and Jeffrey Nichols. Trailblazer: enabling blind users to blaze trails through the web. In *Proceedings of the 14th international conference on Intelligent user interfaces*, pages 177–186. ACM, 2009. 3.4

[48] Jeffrey P Bigham, Jeremy T Brudvik, and Bernie Zhang. Accessibility by demonstration: enabling end users to guide developers to web accessibility solutions. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*, pages 35–42. ACM, 2010. 3.4

[49] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 7.4.4

[50] Paul Brie, Nicolas Burny, Arthur Sluÿters, and Jean Vanderdonckt. Evaluating a large language model on searching for gui layouts. *Proceedings of the ACM on Human-Computer Interaction*, 7(EICS):1–37, 2023. 2.2.3

[51] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. 8.5.5

[52] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023. 8.2, 8.5.5

[53] Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. Vins: Visual search for mobile user interface design. In *Proceedings of the 2021*

CHI Conference on Human Factors in Computing Systems, pages 1–14, 2021. 4.4.1, 4.4.1, 6.4, 9.3, 9.3.2, 9.5.2, 9.6.3

[54] Andrea Burns, Deniz Arsan, Sanjna Agrawal, Ranjitha Kumar, Kate Saenko, and Bryan A Plummer. Interactive mobile app navigation with uncertain or under-specified natural language commands. *arXiv preprint arXiv:2202.02312*, 2022. 4.2, 4.4.3

[55] Bill Buxton. *Sketching user experiences: getting the design right and the right design*. Morgan kaufmann, 2010. 1

[56] Francisco M Castro, Manuel J Marín-Jiménez, Nicolás Guil, Cordelia Schmid, and Karteek Alahari. End-to-end incremental learning. In *Proceedings of the European conference on computer vision (ECCV)*, pages 233–248, 2018. 2.2.2, 6.5.3

[57] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. 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*, pages 245–256, 2011. 2.2.2, 4.3.2, 5.7.3, 5.8

[58] Youli Chang, Sehi L'Yi, Kyle Koh, and Jinwook Seo. Understanding users' touch behavior on large mobile touch-screens and assisted targeting by tilting gesture. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1499–1508, 2015. 7.4.1

[59] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009. 4.4.2

[60] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002. 4.6

[61] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 665–676, 2018. 2.2.2, 2.2.3, 5.3.2, 5.7.3

[62] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 322–334. IEEE, 2020. 2.2.2, 2.2.2, 6.6

[63] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020. 4.4.1

[64] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. Towards complete icon labeling in mobile applications. In *CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2022. 2.2.2, 4.5.1, 6.6

[65] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto,

Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 8.3, 8.5.1, 8.5.2, 8.6, 9.3.2

[66] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. Storydroid: Automated generation of storyboard for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 596–607. IEEE, 2019. 6.4.3

[67] Chin-Yi Cheng, Forrest Huang, Gang Li, and Yang Li. Play: Parametrically conditioned layout generation using latent diffusion. *arXiv preprint arXiv:2301.11529*, 2023. 2.2.3

[68] Yifei Cheng, Yukang Yan, Xin Yi, Yuanchun Shi, and David Lindlbauer. Semanticadapt: Optimization-based adaptation of mixed reality layouts leveraging virtual-physical semantic connections. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 282–297, 2021. 2.1.1

[69] Mehdi Cherti, Romain Beaumont, Ross Wightman, Mitchell Wortsman, Gabriel Ilharco, Cade Gordon, Christoph Schuhmann, Ludwig Schmidt, and Jenia Jitsev. Reproducible scaling laws for contrastive language-image learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2818–2829, 2023. 9.4.1, 9.5.1

[70] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL `https://lmsys.org/blog/2023-03-30-vicuna/`. 9.5.1

[71] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023. 9.3.1

[72] Elizabeth Clark, Tal August, Sofia Serrano, Nikita Haduong, Suchin Gururangan, and Noah A. Smith. All that's 'human' is not gold: Evaluating human evaluation of generated text. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 7282–7296, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.565. URL `https://aclanthology.org/2021.acl-long.565`. 6.5.2

[73] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 957–969. IEEE, 2021. 6.4.3

[74] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 957–969. IEEE, 2021. 4.4.3, 4.4.3, 4.4.3

[75] Allen Cypher. Eager: Programming repetitive tasks by example. In *Readings in human–computer interaction*, pages 804–810. Elsevier, 1995. 3.4

[76] Melissa Dawe. Desperately seeking simplicity: How young adults with cognitive disabilities and their families adopt assistive technologies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 1143–1152, New York, NY, USA, 2006. ACM. ISBN 1-59593-372-7. doi: 10.1145/1124772.1124943. URL `http://doi.acm.org/10.1145/1124772.1124943`. 2.1.2

[77] Niraj Ramesh Dayama, Kashyap Todi, Taru Saarelainen, and Antti Oulasvirta. Grids: Interactive layout design with integer programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020. 2.2.3

[78] Lilian de Greef, Mayank Goel, Min Joon Seo, Eric C. Larson, James W. Stout, James A. Taylor, and Shwetak N. Patel. Bilicam: Using mobile phones to monitor newborn jaundice. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 331–342, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2968-2. doi: 10.1145/2632048.2632076. URL `http://doi.acm.org/10.1145/2632048.2632076`. 2.2.1

[79] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. Erica: Interaction mining mobile apps. In *Proceedings of the 29th annual symposium on user interface software and technology*, pages 767–776, 2016. 4.5.1

[80] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*, pages 845–854, 2017. 2.2.2, 4.2, 4.3.2, 4.4.2, 4.4.3, 4.5.1, 4.5.2, 5.5.1, 8.5.3, 9.3, 9.6.3, 9.7.2

[81] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, page 845–854, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349819. doi: 10.1145/3126594.3126651. URL `https://doi.org/10.1145/3126594.3126651`. 2.2.2, 6.2, 6.4

[82] Biplab Deka, Bardia Doosti, Forrest Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Ranjitha Kumar, Tao Dong, and Jeffrey Nichols. An early rico retrospective: Three years of uses for a mobile app dataset. In *Artificial Intelligence for Human Computer Interaction: A Modern Approach*, pages 229–256. Springer, 2021. 4.5.1

[83] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023. 8.3.2, 8.3.4

[84] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 2.2.2

[85] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human–*

Computer Interaction, 16(2-4):97–166, 2001. 2.1.1

[86] Morgan Dixon and James Fogarty. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1525–1534, 2010. 2.2.2

[87] Morgan Dixon and James Fogarty. Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, page 1525–1534, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605589299. doi: 10.1145/1753326.1753554. URL `https://doi.org/10.1145/1753326.1753554`. 2.1.2, 7.3

[88] Jesse Dodge, Maarten Sap, Ana Marasović, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. *arXiv preprint arXiv:2104.08758*, 2021. 9.3.1

[89] Samuel Dodge and Lina Karam. Understanding how image quality affects deep neural networks. In *2016 eighth international conference on quality of multimedia experience (QoMEX)*, pages 1–6. IEEE, 2016. 4.3.1

[90] Peitong Duan, Casimir Wierzynski, and Lama Nachman. Optimizing user interface layouts via gradient descent. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020. 2.2.3, 7.4.3

[91] Peitong Duan, Jeremy Warner, Yang Li, and Bjoern Hartmann. Generating automatic feedback on ui mockups with large language models. *arXiv preprint arXiv:2403.13139*, 2024. 2.2.3, 9.5.2, 9.7.3

[92] AE Elo. The proposed uscf rating system, its development, theory, and applications. chess life xxii (8): 242–247, 1967. 8.5.2

[93] Katherine Eng, Richard L Lewis, Irene Tollinger, Alina Chu, Andrew Howes, and Alonso Vera. Generating automated predictions of behavior strategically adapted to specific performance objectives. In *Proceedings of the sigchi conference on human factors in computing systems*, pages 621–630, 2006. 7.2

[94] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996. 8.3.3, 9.3.2

[95] João Marcelo Evangelista Belo, Mathias N Lystbæk, Anna Maria Feit, Ken Pfeuffer, Peter Kán, Antti Oulasvirta, and Kaj Grønbæk. Auit–the adaptive user interfaces toolkit for designing xr applications. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, pages 1–16, 2022. 2.1.1

[96] Sarah P Everett and Michael D Byrne. Unintended effects: Varying icon spacing changes users' visual search strategy. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 695–702, 2004. 7.2

[97] WA Falcon and .al. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3, 2019. 5.10

[98] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023. 8.4

[99] Anna Maria Feit, Lukas Vordemann, Seonwook Park, Caterina Berube, and Otmar Hilliges. Detecting relevance during decision-making from eye movements for ui adaptation. In *ACM Symposium on Eye Tracking Research and Applications*, pages 1–11, 2020. 2.1.1

[100] Shirin Feiz, Jason Wu, Xiaoyi Zhang, Amanda Swearngin, Titus Barik, and Jeffrey Nichols. Understanding screen relationships from screenshots of smartphone applications. In *27th International Conference on Intelligent User Interfaces*, pages 447–458, 2022. 4.4.3, 4.4.3, 8.3.1, 8.6.2

[101] Shirin Feiz, Jason Wu, Xiaoyi Zhang, Amanda Swearngin, Titus Barik, and Jeffrey Nichols. Understanding screen relationships from screenshots of smartphone applications. In *27th International Conference on Intelligent User Interfaces*, IUI ’22, page 447–458, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391443. doi: 10.1145/3490099.3511109. URL `https://doi.org/10.1145/3490099.3511109`. 2.2.2, 6.3.2

[102] Sidong Feng, Suyu Ma, Jinzhong Yu, Chunyang Chen, TingTing Zhou, and Yankun Zhen. Auto-icon: An automated code generation tool for icon designs assisting in ui development. In *26th International Conference on Intelligent User Interfaces*, pages 59–69, 2021. 2.2.2

[103] Andreas Fischer, Kaspar Riesen, and Horst Bunke. Improved quadratic time approximation of graph edit distance by combining hausdorff matching and greedy assignment. *Pattern Recognition Letters*, 87:55–62, 2017. 5.6.1

[104] Michael Fischer, Giovanni Campagna, Silei Xu, and Monica S Lam. Brassau: automatic generation of graphical user interfaces for virtual assistants. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 1–12, 2018. 7.7.3

[105] Raymond Fok, Mingyuan Zhong, Anne Spencer Ross, James Fogarty, and Jacob O Wobbrock. A large-scale longitudinal analysis of missing label accessibility failures in android apps. In *CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2022. 4.5.1

[106] Rachel L. Franz, Jacob O. Wobbrock, Yi Cheng, and Leah Findlater. Perception and adoption of mobile accessibility features by older adults experiencing ability changes. In *Proceedings of the 2019 SIGACCESS Conference on Accessibility and Computing (ASSETS ’19)*, 2019. 3.2

[107] Jingwen Fu, Xiaoyi Zhang, Yuwang Wang, Wenjun Zeng, Sam Yang, and Grayson Hilliard. Understanding mobile gui: from pixel-words to screen-sentences. *arXiv preprint arXiv:2105.11941*, 2021. 2.2.2

[108] Samir Yitzhak Gadre, Gabriel Ilharco, Alex Fang, Jonathan Hayase, Georgios Smyrnis,

Thao Nguyen, Ryan Marten, Mitchell Wortsman, Dhruba Ghosh, Jieyu Zhang, et al. Datacomp: In search of the next generation of multimodal datasets. *Advances in Neural Information Processing Systems*, 36, 2024. 9.5.1

[109] Krzysztof Gajos and Daniel S. Weld. Supple: Automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, IUI '04, pages 93–100, New York, NY, USA, 2004. ACM. ISBN 1-58113-815-6. doi: 10.1145/964442.964461. URL `http://doi.acm.org/10.1145/964442.964461`. 3.4

[110] Krzysztof Gajos and Daniel S Weld. Supple: automatically generating user interfaces. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 93–100, 2004. 2.1.1, 2.2.2, 3.1, 9.2

[111] Krzysztof Gajos and Daniel S Weld. Preference elicitation for interface optimization. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 173–182, 2005. 2.2.3, 9.2, 9.3, 9.7.1, 9.7.2

[112] Krzysztof Z Gajos, Jacob O Wobbrock, and Daniel 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*, pages 231–240, 2007. 9.2

[113] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1257–1266, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357250. URL `http://doi.acm.org/10.1145/1357054.1357250`. 3.4

[114] Krzysztof Z Gajos, Jacob O Wobbrock, and Daniel S Weld. Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 1257–1266, 2008. 2.1.1

[115] Krzysztof Z Gajos, Daniel S Weld, and Jacob O Wobbrock. Automatically generating personalized user interfaces with supple. *Artificial Intelligence*, 174(12-13):910–950, 2010. 2.1.1

[116] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The journal of machine learning research*, 17(1):2096–2030, 2016. 4.4.3

[117] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020. 4.2

[118] George Giannakopoulos and Vangelis Karkaletsis. Autosummeng and memog in evaluating guided summaries. In *TAC*, 2011. 2.2.3

[119] Camille Gobert, Kashyap Todi, Gilles Bailly, and Antti Oulasvirta. Sam: a modular frame-

work for self-adapting web menus. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pages 481–484, 2019. 2.1.1

[120] Gabriel Goh, Nick Cammarata, Chelsea Voss, Shan Carter, Michael Petrov, Ludwig Schubert, Alec Radford, and Chris Olah. Multimodal neurons in artificial neural networks. *Distill*, 6(3):e30, 2021. 8.3.3

[121] Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, pages 959–976, 2012. 5.5.2, 5.6.2

[122] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 2.2.2, 6.5.1

[123] Kelley Gordon. 5 principles of visual design in ux. `https://www.nngroup.com/articles/principles-visual-design/`, 2020. Accessed: 2024-03-25. 9.3.1

[124] Wayne D Gray and Deborah A Boehm-Davis. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of experimental psychology: applied*, 6(4):322, 2000. 7.2, 7.7.3

[125] W3C Working Group. Introduction to model-based user interfaces, 2014. URL `https://www.w3.org/TR/mbui-intro/`. 2.2.2, 5.3.1

[126] Gianluigi Grzincich, Rolando Gagliardini, Anna Bossi, Sergio Bella, Giuseppe Cimino, Natalia Cirilli, Laura Viviani, Eugenia Iacinti, and Serena Quattrucci. Evaluation of a home telemonitoring service for adult patients with cystic fibrosis: a pilot study. *Journal of telemedicine and telecare*, 16(7):359–362, 2010. 2.2.1

[127] Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, et al. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*, 2023. 9.6.1

[128] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023. 8.2

[129] Anhong Guo, Junhan Kong, Michael Rivera, Frank F Xu, and Jeffrey P Bigham. Statelens: A reverse engineering solution for making existing dynamic touchscreens accessible. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 371–385, 2019. 3.4

[130] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006. 6.4.3

[131] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006. 4.4.3, 9.4.1

[132] Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020. 5.4.2

[133] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for

image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 6.3.2

[134] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 4.4.2, 4.4.3, 5.4.1

[135] Zecheng He, Srinivas Sunkara, Xiaoxue Zang, Ying Xu, Lijuan Liu, Nevan Wichers, Gabriel Schubiner, Ruby Lee, and Jindong Chen. Actionbert: Leveraging user actions for semantic understanding of user interfaces. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7):5931–5938, May 2021. doi: 10.1609/aaai.v35i7.16741. URL `https://ojs.aaai.org/index.php/AAAI/article/view/16741`. 2.2.2

[136] Jack Hessel, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. Clipscore: A reference-free evaluation metric for image captioning. *arXiv preprint arXiv:2104.08718*, 2021. 2.2.3, 8.5.2

[137] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017. 2.2.3

[138] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 5.4.2

[139] Christian Holz and Patrick Baudisch. The generalized perceived input point model and how to double touch accuracy by extracting fingerprints. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 581–590, 2010. 2.2.1

[140] Christian Holz and Patrick Baudisch. Understanding touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2501–2510, 2011. 2.2.1

[141] Eric Horvitz. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 159–166. ACM, 1999. 3.7

[142] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. 6.3.2

[143] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 8.4

[144] Yushi Hu, Benlin Liu, Jungo Kasai, Yizhong Wang, Mari Ostendorf, Ranjay Krishna, and Noah A Smith. Tifa: Accurate and interpretable text-to-image faithfulness evaluation with question answering. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 20406–20417, 2023. 2.2.3

[145] Forrest Huang, John F Canny, and Jeffrey Nichols. Swire: Sketch-based user interface retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–10, 2019. 9.5.2

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

Systems, CHI '19, page 1–10, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359702. doi: 10.1145/3290605.3300334. URL `https://doi.org/10.1145/3290605.3300334`. 2.2.2

[147] Forrest Huang, Gang Li, Xin Zhou, John F Canny, and Yang Li. Creating user interface mock-ups from high-level text descriptions with deep-learning models. *arXiv preprint arXiv:2110.07775*, 2021. 2.2.3

[148] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016. 4.4.2

[149] Charlotte Humphrey, Katia Gilhome Herbst, and Shaista Faurqi. Some characteristics of the hearing-impaired elderly who do not present themselves for rehabilitation. *British Journal of Audiology*, 15(1):25–30, 1981. 3.6.4

[150] Amy Hurst, Krzysztof Gajos, Leah Findlater, Jacob Wobbrock, Andrew Sears, and Shari Trewin. Dynamic accessibility: Accommodating differences in ability and situation. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 41–44, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0268-5. doi: 10.1145/1979742.1979589. URL `http://doi.acm.org/10.1145/1979742.1979589`. 3.4

[151] Gabriel Ilharco, Mitchell Wortsman, Ross Wightman, Cade Gordon, Nicholas Carlini, Rohan Taori, Achal Dave, Vaishaal Shankar, Hongseok Namkoong, John Miller, Hannaneh Hajishirzi, Ali Farhadi, and Ludwig Schmidt. Openclip, July 2021. URL `https://doi.org/10.5281/zenodo.5143773`. If you use this software, please cite it as below. 8.3.3, 8.4, 8.5.2

[152] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)*, pages 1681–1691, 2015. 5.4.3

[153] Bernard J Jansen. The graphical user interface. *ACM SIGCHI Bulletin*, 30(2):22–26, 1998. 2.2.3

[154] Fred Jelinek, Robert L Mercer, Lalit R Bahl, and James K Baker. Perplexity—a measure of the difficulty of speech recognition tasks. *The Journal of the Acoustical Society of America*, 62(S1):S63–S63, 1977. 2.2.3

[155] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023. 9.3.1

[156] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024. 9.3.2

[157] Yue Jiang, Wolfgang Stuerzlinger, and Christof Lutteroth. Reverseorc: Reverse engineer-

ing of resizable user interface layouts with or-constraints. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–18, 2021. 2.1.2, 7.7.3

[158] Dan Jurafsky and James H. Martin. *Speech & language processing 3rd ed. draft*. 2020. 5.6.1

[159] Miles A Kimball. Visual design principles: An empirical study of design lore. *Journal of Technical Writing and Communication*, 43(1):3–41, 2013. 9.3.2

[160] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 7.4.3

[161] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017. 2.2.2, 6.5.3

[162] Boris Knyazev, Harm de Vries, Cătălina Cangea, Graham W Taylor, Aaron Courville, and Eugene Belilovsky. Graph density-aware losses for novel compositions in scene graph generation. *arXiv preprint arXiv:2005.08230*, 2020. 2.2.2

[163] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022. 9.5.1

[164] Kristian Kolthoff, Christian Bartelt, and Simone Paolo Ponzetto. Gui2wire: rapid wireframing with a mined and large-scale gui repository using natural language requirements. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1297–1301, 2020. 2.2.3

[165] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, et al. Openassistant conversations–democratizing large language model alignment. *arXiv preprint arXiv:2304.07327*, 2023. 8.3.1

[166] Sarah Krings, Enes Yigitbas, Ivan Jovanovikj, Stefan Sauer, and Gregor Engels. Development framework for context-aware augmented reality applications. In *Companion Proceedings of the 12th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 1–6, 2020. 2.1.1

[167] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A Shamma, et al. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International journal of computer vision*, 123(1):32–73, 2017. 5.3.2

[168] Rebecca Krosnick, Sang Won Lee, Walter S Laseck, and Steve Onev. Expresso: Building responsive interfaces with keyframes. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 39–47. IEEE, 2018. 7.7.3

[169] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955. 5.5.1

[170] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad,

Scott R Klemmer, and Jerry O Talton. Webzeitgeist: design mining the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3083–3092, 2013. 4.2, 8.3.1, 9.3.1, 9.6.3

[171] Konstantin Kuznetsov, Chen Fu, Song Gao, David N. Jansen, Lijun Zhang, and Andreas Zeller. Frontmatter: Mining android user interfaces at scale. 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 1580–1584, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3473125. URL https://doi.org/10.1145/3468264.3473125. 2.2.2, 6.2, 6.4

[172] Daniël Lakens. Calculating and reporting effect sizes to facilitate cumulative science: a practical primer for t-tests and anovas. *Frontiers in psychology*, 4:863, 2013. 7.5.2

[173] Xuan Nhat Lam, Thuc Vu, Trong Duc Le, and Anh Duc Duong. Addressing cold-start problem in recommendation systems. In *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pages 208–211. ACM, 2008. 3.2

[174] James A Landay. Silk: sketching interfaces like krazy. In *Conference companion on Human factors in computing systems*, pages 398–399, 1996. 2.2.2, 2.2.3

[175] Eric C Larson, Mayank Goel, Gaetano Boriello, Sonya Heltshe, Margaret Rosenfeld, and Shwetak N Patel. Spirosmart: using a microphone to measure lung function on a mobile phone. In *Proceedings of the 2012 ACM Conference on ubiquitous computing*, pages 280–289. ACM, 2012. 2.2.1

[176] Huy Viet Le, Sven Mayer, Patrick Bader, and Niels Henze. Fingers' range and comfortable area for one-handed smartphone interaction beyond the touchscreen. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018. 7.4.1

[177] Huy Viet Le, Sven Mayer, Benedict Steuerlein, and Niels Henze. Investigating unintended inputs for one-handed touch interaction beyond the touchscreen. In *Proceedings of the 21st International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 1–14, 2019. 2.2.1

[178] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. 2022. 9.3.1

[179] Colin Lea, Vikramjit Mitra, Aparna Joshi, Sachin Kajarekar, and Jeffrey P. Bigham. Sep-28k: A dataset for stuttering event detection from podcasts with people who stutter. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2021)*, 2021. 3.2

[180] Kenton Lee, Mandar Joshi, Iulia Raluca Turc, Hexiang Hu, Fangyu Liu, Julian Martin Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. Pix2struct: Screenshot parsing as pretraining for visual language understanding. In *International Conference on Machine Learning*, pages 18893–18912. PMLR, 2023. 2.2.3, 9.3.1, 9.3.2, 9.4.2

[181] Luis A Leiva, Asutosh Hota, and Antti Oulasvirta. Enrico: A dataset for topic modeling of mobile ui designs. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 1–4, 2020. 2.2.2, 4.4.2, 4.4.2, 4.4.2, 9.3

[182] Luis A. Leiva, Asutosh Hota, and Antti Oulasvirta. Enrico: A dataset for topic modeling of mobile ui designs. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '20, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380522. doi: 10.1145/3406324.3410710. URL `https://doi.org/10.1145/3406324.3410710`. 6.4.1

[183] Luis A Leiva, Asutosh Hota, and Antti Oulasvirta. Describing ui screenshots in natural language. *ACM Transactions on Intelligent Systems and Technology*, 14(1):1–28, 2022. 8.3.1

[184] Gang Li and Yang Li. Spotlight: Mobile ui understanding using vision-language models with a focus. *arXiv preprint arXiv:2209.14927*, 2022. 9.7.3

[185] Gang Li, Gilles Baechler, Manuel Tragut, and Yang Li. Learning to denoise raw mobile ui layouts for improving datasets at scale. In *CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2022. 4.3.2

[186] Gang Li, Gilles Baechler, Manuel Tragut, and Yang Li. Learning to denoise raw mobile ui layouts for improving datasets at scale. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391573. doi: 10.1145/3491102.3502042. URL `https://doi.org/10.1145/3491102.3502042`. 6.4.1

[187] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. Layoutgan: Generating graphic layouts with wireframe discriminators. *arXiv preprint arXiv:1901.06767*, 2019. 2.2.3

[188] Jianan Li, Jimei Yang, Jianming Zhang, Chang Liu, Christina Wang, and Tingfa Xu. Attribute-conditioned layout gan for automatic graphic design. *IEEE Transactions on Visualization and Computer Graphics*, 27(10):4039–4048, 2020. 2.2.3

[189] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*, 2023. 8.3.1

[190] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. 8.2, 8.3.2, 8.3.3, 8.5.5

[191] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. Sugilite: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 6038–6049, 2017. 4.2, 5.8

[192] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. Screen2vec: Semantic embedding of gui screens and gui components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021. 2.2.2

[193] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. Screen2vec: Semantic embedding of gui screens and gui components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380966. doi: 10.1145/3411764.3445049. URL `https://doi.org/10.1145/3411764.3445049`. 2.2.2

[194] Yang Li, Samy Bengio, and Gilles Bailly. Predicting human performance in vertical menu selection using deep learning. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2018. 7.4.3

[195] Yang Li, Julien Amelot, Xin Zhou, Samy Bengio, and Si Si. Auto completion of user interface layout design using transformer-based tree decoders. *arXiv preprint arXiv:2001.05308*, 2020. 2.2.3

[196] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. Mapping natural language instructions to mobile ui action sequences. *arXiv preprint arXiv:2005.03776*, 2020. 2.2.2, 4.2

[197] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. Widget captioning: Generating natural language description for mobile user interface elements. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5495–5510, 2020. 2.2.2

[198] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017. 6.4.3

[199] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017. 4.4.3, 6.4

[200] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1070–1073. IEEE, 2019. 6.4.3

[201] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1070–1073. IEEE, 2019. 4.4.3

[202] Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017. 2.2.2, 6.5.3

[203] Paul Pu Liang, Yiwei Lyu, Xiang Fan, Zetian Wu, Yun Cheng, Jason Wu, Leslie Chen, Peter Wu, Michelle A Lee, Yuke Zhu, et al. Multibench: Multiscale benchmarks for multimodal representation learning. *arXiv preprint arXiv:2107.07502*, 2021. 4.4.2

[204] Yi-Chi Liao, Kashyap Todi, Aditya Acharya, Antti Keurulainen, Andrew Howes, and Antti Oulasvirta. Rediscovering affordance: A reinforcement learning perspective. In

Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, pages 1–15, 2022. 8.6.1

[205] Yi-Chi Liao, Kashyap Todi, Aditya Acharya, Antti Keurulainen, Andrew Howes, and Antti Oulasvirta. Rediscovering affordance: A reinforcement learning perspective. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391573. doi: 10.1145/3491102.3501992. URL `https://doi.org/10.1145/3491102.3501992`. 2.2.2

[206] William Lidwell, Kritina Holden, and Jill Butler. *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010. 9.3.1

[207] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023. 8.6.1

[208] Brian Y Lim and Anind K Dey. Toolkit to support intelligibility in context-aware applications. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, pages 13–22, 2010. 2.1.1

[209] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004. 2.2.3

[210] David Lindlbauer, Anna Maria Feit, and Otmar Hilliges. Context-aware online adaptation of mixed reality interfaces. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*, pages 147–160, 2019. 2.1.1

[211] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning. *arXiv preprint arXiv:2310.03744*, 2023. 9.4, 9.5.2

[212] Haotian Liu, Chunyuan Li, Yuheng Li, Bo Li, Yuanhan Zhang, Sheng Shen, and Yong Jae Lee. Llava-next: Improved reasoning, ocr, and world knowledge, January 2024. URL `https://llava-vl.github.io/blog/2024-01-30-llava-next/`. 9.5.1

[213] Thomas F Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning design semantics for mobile apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 569–579, 2018. 2.2.2, 6.6

[214] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016. 4.4.1

[215] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023. 8.5.5

[216] Kurt Luther, Jari-Lee Tolentino, Wei Wu, Amy Pavel, Brian P Bailey, Maneesh Agrawala, Björn Hartmann, and Steven P Dow. Structuring, aggregating, and evaluating crowdsourced design critique. In *Proceedings of the 18th ACM conference on computer supported cooperative work & social computing*, pages 473–485, 2015. 9.3.1

[217] Christof Lutteroth. Automated reverse engineering of hard-coded gui layouts. In *Proceedings of the ninth conference on Australasian user interface-Volume 76*, pages 65–73, 2008. 7.7.3

[218] Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, and Eduard Hovy. Stack-pointer networks for dependency parsing. *arXiv preprint arXiv:1805.01087*, 2018. 5.4.2, 5.6

[219] I Scott MacKenzie. Fitts' law as a research and design tool in human-computer interaction. *Human-computer interaction*, 7(1):91–139, 1992. 2.2.1, 7.4.1

[220] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024. 9.6.1

[221] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens Van Der Maaten. Exploring the limits of weakly supervised pretraining. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 181–196, 2018. 5.5.2

[222] Jennifer Mankoff, Anind K Dey, Gary Hsieh, Julie Kientz, Scott Lederer, and Morgan Ames. Heuristic evaluation of ambient displays. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 169–176, 2003. 7.6.1

[223] Sven Mayer, Huy Viet Le, Markus Funk, and Niels Henze. Finding the sweet spot: Analyzing unrestricted touchscreen interaction in-the-wild. In *Proceedings of the 2019 ACM International Conference on Interactive Surfaces and Spaces*, pages 171–179, 2019. 2.2.1, 7.4.1

[224] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018. 5.7.1

[225] Forough Mehralian, Navid Salehnamadi, and Sam Malek. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 107–118, 2021. 2.2.2

[226] J-L Meunier. Optimized xy-cut for determining a page reading order. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 347–351. IEEE, 2005. (document), 5.7.2, 5.8

[227] Aliaksei Miniukovich and Antonella De Angeli. Computation of interface aesthetics. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1163–1172, 2015. 2.2.3, 9.2

[228] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, B. Yang, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel, J. Krishnamurthy, N. Lao, K. Mazaitis, T. Mohamed, N. Nakashole, E. Platanios, A. Ritter, M. Samadi, B. Settles, R. Wang, D. Wijaya, A. Gupta, X. Chen, A. Saparov, M. Greaves, and J. Welling. Never-ending learning.

Commun. ACM, 61(5):103–115, apr 2018. ISSN 0001-0782. doi: 10.1145/3191513. URL `https://doi.org/10.1145/3191513`. 2.2.2, 6.2

[229] Tom Mitchell, William Cohen, Estevam Hruschka, Partha Talukdar, Bishan Yang, Justin Betteridge, Andrew Carlson, Bhavana Dalvi, Matt Gardner, Bryan Kisiel, et al. Never-ending learning. *Communications of the ACM*, 61(5):103–115, 2018. 4.5.2

[230] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. Automated reporting of gui design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*, pages 165–175, 2018. 4.5.1, 9.2

[231] Giulio Mori, Fabio Paterno, and Carmen Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering*, 30(8):507–520, 2004. 2.1.1

[232] Martez E Mott and Jacob O Wobbrock. Cluster touch: Improving touch accuracy on smartphones for people with motor and situational impairments. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2019. 7.4.1

[233] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023. 8.2, 8.5.4, 8.5.5

[234] Michael Nebeling, Maximilian Speicher, and Moira Norrie. W3touch: metrics-based web page adaptation for touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2311–2320, 2013. 2.1.1

[235] Matei Negulescu and Joanna McGrenere. Grip change as an information side channel for mobile touch interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1519–1522, 2015. 7.4.1

[236] Timothy Nguyen, Zhourong Chen, and Jaehoon Lee. Dataset meta-learning from kernel ridge-regression. *arXiv preprint arXiv:2011.00050*, 2020. 2.2.2, 6.4.3, 6.5.3

[237] Timothy Nguyen, Roman Novak, Lechao Xiao, and Jaehoon Lee. Dataset distillation with infinitely wide convolutional networks. *Advances in Neural Information Processing Systems*, 34:5186–5198, 2021. 2.2.2, 6.4.3, 6.5.3

[238] Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–259. IEEE, 2015. 2.2.2, 5.6, 5.7.3

[239] Jeffrey Nichols, Brad A Myers, Michael Higgins, Joseph Hughes, Thomas K Harris, Roni Rosenfeld, and Mathilde Pignol. Generating remote control interfaces for complex appliances. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 161–170, 2002. 2.2.2, 2.2.3

[240] Jeffrey Nichols, Zhigang Hua, and John Barton. Highlight: a system for creating and deploying mobile web applications. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 249–258, 2008. 7.7.3

[241] Jakob Nielsen. Finding usability problems through heuristic evaluation. In *Proceedings*

of the SIGCHI conference on Human factors in computing systems, pages 373–380, 1992. 9.2

[242] Jakob Nielsen. *Heuristic Evaluation*, page 25–62. John Wiley & Sons, Inc., USA, 1994. ISBN 0471018775. 7.6.1

[243] Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 152–158, 1994. 2.2.3

[244] Donald A Norman. *The psychology of everyday things.* Basic books, 1988. 2.2.2

[245] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. 8.5.5

[246] Antti Oulasvirta, Samuli De Pascale, Janin Koch, Thomas Langerak, Jussi Jokinen, Kashyap Todi, Markku Laine, Manoj Kristhombuge, Yuxi Zhu, Aliaksei Miniukovich, et al. Aalto interface metrics (aim) a service and codebase for computational gui evaluation. In *Adjunct Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 16–19, 2018. 2.2.3, 8.5.2, 9.2, 9.5.1

[247] Antti Oulasvirta, Niraj Ramesh Dayama, Morteza Shiripour, Maximilian John, and Andreas Karrenbauer. Combinatorial optimization of graphical user interface designs. *Proceedings of the IEEE*, 108(3):434–464, 2020. 8.5.2

[248] Antti Oulasvirta, Jussi P. P. Jokinen, and Andrew Howes. Computational rationality as a theory of interaction. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391573. doi: 10.1145/3491102.3517739. URL `https://doi.org/10.1145/3491102.3517739`. 2.2.2

[249] Antti Oulasvirta, Jussi PP Jokinen, and Andrew Howes. Computational rationality as a theory of interaction. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2022. 9.2

[250] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022. (document), 8.1, 8.3, 8.3.4, 8.6.2, 9.6.1

[251] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002. 2.2.3

[252] Seonwook Park, Christoph Gebhardt, Roman Rädle, Anna Maria Feit, Hana Vrzakova, Niraj Ramesh Dayama, Hui-Shyong Yeo, Clemens N Klokmose, Aaron Quigley, Antti Oulasvirta, et al. Adam: Adapting multi-user interfaces for collaborative environments in real-time. In *Proceedings of the 2018 CHI conference on human factors in computing systems*, pages 1–14, 2018. 2.1.1

[253] Mihir Parmar, Swaroop Mishra, Mor Geva, and Chitta Baral. Don't blame the annotator: Bias already starts in the annotation instructions, 2022. URL `https://arxiv.org/`

abs/2205.00415. 6.5.2

[254] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 5.10

[255] Fabio Paterno', Carmen Santoro, and Lucio Davide Spano. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4):1–30, 2009. 2.1.1

[256] John C Platt and Alan H Barr. Constrained differential optimization for neural networks. 1988. 7.5.3

[257] Angel R Puerta. A model-based interface development environment. *IEEE Software*, 14 (4):40–47, 1997. 2.2.2, 5.3.1

[258] Angel R Puerta and David Maulsby. Mobi-d: a model-based development environment for user-centered design. In *CHI'97 Extended Abstracts on Human Factors in Computing Systems*, pages 4–5. 1997. 2.2.3

[259] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018. 9.4.2

[260] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021. 8.2, 8.6.1, 9.2, 9.4, 9.4.1, 9.4.1, 9.9

[261] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023. 8.3.4

[262] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21 (140):1–67, 2020. URL http://jmlr.org/papers/v21/20-074.html. 9.3.1

[263] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2001–2010, 2017. 2.2.2, 6.5.3

[264] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019. 9.3.2

[265] Luz Rello, Miguel Ballesteros, Abdullah Ali, Miquel Serra, D Alaron, and Jeffrey P Bigham. Dytective: Diagnosing risk of dyslexia with a game. In *Proceedings of Pervasive Health*, volume 16, 2016. URL http://www.luzrello.com/Publications_files/PerHealth2016-Dytective.pdf. 2.2.1

[266] Luz Rello, Miguel Ballesteros, Abdullah Ali, Miquel Serra, Daniela Alarcón Sánchez, and Jeffrey P. Bigham. Dytective: Diagnosing risk of dyslexia with a game. In *Proceedings of*

the 10th EAI International Conference on Pervasive Computing Technologies for Healthcare, PervasiveHealth '16, pages 89–96, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-1-63190-051-8. URL `http://dl.acm.org/citation.cfm?id=3021319.3021333`. 2.2.1, 3.2, 3.4

[267] Luz Rello, Ricardo Baeza-Yates, Abdullah Ali, Jeffrey P. Bigham, and Miquel Serra. Predicting risk of dyslexia with an online gamified test. *PLOS ONE*, 15(12):1–15, 12 2020. doi: 10.1371/journal.pone.0241687. URL `https://doi.org/10.1371/journal.pone.0241687`. 3.2

[268] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *arXiv preprint arXiv:1506.01497*, 2015. 5.4.1

[269] Andreas Riegler and Clemens Holzmann. Measuring visual user interface complexity of mobile applications with metrics. *Interacting with Computers*, 30(3):207–223, 2018. 2.2.3

[270] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. Codemend: Assisting interactive programming with bimodal embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 247–258, 2016. 2.2.3

[271] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *Advances in neural information processing systems*, 29, 2016. 2.2.3

[272] Guillaume Sanchez, Honglu Fan, Alexander Spangher, Elad Levi, Pawan Sasanka Ammanamanchi, and Stella Biderman. Stay on topic with classifier-free guidance. *arXiv preprint arXiv:2306.17806*, 2023. 9.6.3

[273] Sayan Sarcar, Jussi PP Jokinen, Antti Oulasvirta, Zhenxin Wang, Chaklam Silpasuwanchai, and Xiangshi Ren. Ability-based optimization of touchscreen interactions. *IEEE Pervasive Computing*, 17(1):15–26, 2018. 2.1.1

[274] Badrul Munir Sarwar, George Karypis, Joseph A Konstan, John Riedl, et al. Item-based collaborative filtering recommendation algorithms. *Www*, 1:285–295, 2001. 3.2

[275] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023. 8.2

[276] Viktor Schlegel, Benedikt Lang, Siegfried Handschuh, and André Freitas. Vajra: step-by-step programming with natural language. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pages 30–39, 2019. 2.2.3

[277] Eldon Schoop, Xin Zhou, Gang Li, Zhourong Chen, Bjoern Hartmann, and Yang Li. Predicting and explaining mobile ui tappability with vision modeling and saliency analysis. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–21, 2022. 2.2.3, 9.7.3

[278] Eldon Schoop, Xin Zhou, Gang Li, Zhourong Chen, Björn Hartmann, and Yang Li. Pre-

dicting and explaining mobile ui tappability with vision modeling and saliency analysis. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–21, 2022. 2.2.2, 2.2.2, 6.2, 6.4.1, 6.4.1, 6.5.2

[279] Richard S. Schwerdtfeger. Making the gui talk, 1991. URL `ftp://service.boulder.ibm.com/sns/sr-os2/sr2doc/guitalk.txt`. 2.1.2, 2.2.2

[280] Alborz Rezazadeh Sereshkeh, Gary Leung, Krish Perumal, Caleb Phillips, Minfan Zhang, Afsaneh Fazly, and Iqbal Mohomed. Vasta: a vision and language-assisted smartphone task automation system. In *Proceedings of the 25th international conference on intelligent user interfaces*, pages 22–32, 2020. 2.2.2

[281] Vinoth Pandian Sermuga Pandian, Sarah Suleri, and Matthias Jarke. Synz: Enhanced synthetic dataset for training ui element detectors. In *26th International Conference on Intelligent User Interfaces-Companion*, pages 67–69, 2021. 4.3.2

[282] Pratyusha Sharma, Tamar Rott Shaham, Manel Baradad, Stephanie Fu, Adrian Rodriguez-Munoz, Shivam Duggal, Phillip Isola, and Antonio Torralba. A vision check-up for language models. *arXiv preprint arXiv:2401.01862*, 2024. 9.6.1

[283] Michael Shilman, Percy Liang, and Paul Viola. Learning nongenerative grammatical models for document analysis. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 962–969. IEEE, 2005. 2.2.2

[284] Morteza Shiripour, Niraj Ramesh Dayama, and Antti Oulasvirta. Grid-based genetic operators for graphical layout generation. *Proceedings of the ACM on Human-Computer Interaction*, 5(EICS):1–30, 2021. 7.2, 7.6.1, 7.6.2

[285] Ben Shneiderman, Catherine Plaisant, Maxine Cohen, Steven Jacobs, Niklas Elmqvist, and Nicholas Diakopoulos. *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education, 2016. 9.2, 9.3.1

[286] Chenglei Si, Yanzhe Zhang, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: How far are we from automating front-end engineering? *arXiv preprint arXiv:2403.03163*, 2024. 9.3

[287] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 4.4.2

[288] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE, 2017. 6.4

[289] Ray Smith. An overview of the tesseract ocr engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)*, volume 2, pages 629–633. IEEE, 2007. 7.4.4

[290] Richard Socher, Cliff Chiung-Yu Lin, Andrew Y Ng, and Christopher D Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011. 2.2.2

[291] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014. 4.4.2

[292] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. User inter-

face façades: towards fully adaptable user interfaces. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 309–318, 2006. 2.1.2, 7.3, 7.4.4, 7.4.4

[293] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, pages 843–852, 2017. 6.5.1

[294] Amanda Swearngin and Yang Li. Modeling mobile interface tappability using crowdsourcing and deep learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2019. 2.2.3, 4.4.1, 4.5.1

[295] Amanda Swearngin and Yang Li. Modeling mobile interface tappability using crowdsourcing and deep learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, page 1–11, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359702. doi: 10.1145/3290605.3300305. URL `https://doi.org/10.1145/3290605.3300305`. 2.2.2, 2.2.2, 6.2, 6.4.1, 6.4.1

[296] Amanda Swearngin, Amy J Ko, and James Fogarty. Genie: Input retargeting on the web through command reverse engineering. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 4703–4714, 2017. 2.1.2

[297] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. Scout: Rapid exploration of interface layout alternatives through high-level design constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020. 2.2.3, 9.2

[298] Amanda Swearngin, Jason Wu, Xiaoyi Zhang, Esteban Gomez, Jen Coughenour, Rachel Stukenborg, Bhavya Garg, Greg Hughes, Adriana Hilliard, Jeffrey P Bigham, et al. Towards automated accessibility report generation for mobile apps. *arXiv preprint arXiv:2310.00091*, 2023. 9.7.3

[299] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019. 6.3.2

[300] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023. 9.5.1

[301] MosaicML NLP Team. Introducing mpt-30b: Raising the bar for open-source foundation models, 2023. URL `www.mosaicml.com/blog/mpt-30b`. Accessed: 2023-06-22. 8.5.4, 8.5.5

[302] Alexandru Telea. An image inpainting technique based on the fast marching method. *Journal of graphics tools*, 9(1):23–34, 2004. 7.4.4

[303] Nandan Thakur, Nils Reimers, Johannes Daxenberger, and Iryna Gurevych. Augmented sbert: Data augmentation method for improving bi-encoders for pairwise sentence scoring tasks. *arXiv preprint arXiv:2010.08240*, 2020. 6.3.2

[304] Feng Tian, Xiangmin Fan, Junjun Fan, Yicheng Zhu, Jing Gao, Dakuo Wang, Xiaojun Bi, and Hongan Wang. What can gestures tell?: Detecting motor impairment in early parkinson's from common touch gestural interactions. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 83:1–83:14, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5970-2. doi: 10.1145/3290605.3300313. URL `http://doi.acm.org/10.1145/3290605.3300313`. 2.2.1

[305] Zhi Tian, Chunhua Shen, Hao Chen, and Tong He. Fcos: Fully convolutional one-stage object detection. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9627–9636, 2019. 4.4.1

[306] Kashyap Todi, Daryl Weir, and Antti Oulasvirta. Sketchplore: Sketch and explore with a layout optimiser. In *Proceedings of the 2016 ACM conference on designing interactive systems*, pages 543–555, 2016. 2.2.3, 7.2, 7.6.1, 7.6.2, 8.5.2, 9.2

[307] Kashyap Todi, Luis A Leiva, Daniel Buschek, Pin Tian, and Antti Oulasvirta. Conversations with guis. In *Designing Interactive Systems Conference 2021*, pages 1447–1457, 2021. 2.2.3

[308] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. 8.3.4, 9.3.1

[309] Daniel Toyama, Philippe Hamel, Anita Gergely, Gheorghe Comanici, Amelia Glaese, Zafarali Ahmed, Tyler Jackson, Shibl Mourad, and Doina Precup. Androidenv: A reinforcement learning platform for android. *arXiv preprint arXiv:2105.13231*, 2021. 6.4

[310] Zhuowen Tu, Xiangrong Chen, Alan L Yuille, and Song-Chun Zhu. Image parsing: Unifying segmentation, detection, and recognition. *International Journal of computer vision*, 63(2):113–140, 2005. 2.2.2

[311] Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. Creating a coding assistant with starcoder. *Hugging Face Blog*, 2023. https://huggingface.co/blog/starchat-alpha. 8.2, 8.3.2, 8.5.5

[312] Daniel Vogel and Patrick Baudisch. Shift: a technique for operating pen-based interfaces using touch. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 657–666, 2007. 2.2.1

[313] Tri Vu, Hoan Tran, Kun Woo Cho, Chen Song, Feng Lin, Chang Wen Chen, Michelle Hartley-McAndrew, Kathy Ralabate Doody, and Wenyao Xu. Effective and efficient visual stimuli design for quantitative autism screening: An exploratory study. In *2017 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI)*, pages 297–300. IEEE, 2017. 2.2.1, 3.2

[314] Morten Wahrendorf, Jan D Reinhardt, and Johannes Siegrist. Relationships of disability with age among adults aged 50 to 85: evidence from the united states, england and continental europe. *PloS one*, 8(8):e71893, 2013. 3.6.2

[315] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. Screen2words: Automatic mobile ui summarization with multimodal learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 498–510, 2021. 2.2.2, 8.3.1, 8.5.1, 8.5.3, 8.6.1, 9.3, 9.3.1, 9.7.1

[316] Edward Jay Wang, William Li, Doug Hawkins, Terry Gernsheimer, Colette Norby-Slycord, and Shwetak N. Patel. Hemaapp: Noninvasive blood screening of hemoglobin using smartphone cameras. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, pages 593–604, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4461-6. doi: 10.1145/2971648.2971653. URL `http://doi.acm.org/10.1145/2971648.2971653`. 2.2.1

[317] Jianyi Wang, Kelvin CK Chan, and Chen Change Loy. Exploring clip for assessing the look and feel of images. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 2555–2563, 2023. 2.2.3

[318] Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A Efros. Dataset distillation. *arXiv preprint arXiv:1811.10959*, 2018. 2.2.2, 6.4.3, 6.5.3

[319] Daniel Weld, Corin Anderson, Pedro Domingos, Oren Etzioni, Krzysztof Z Gajos, Tessa Lau, and Steve Wolfman. Automatically personalizing user interfaces. 2003. 2.2.3

[320] Tilo Westermann, Sebastian Möller, and Ina Wechsung. Assessing the relationship between technical affinity, stress and notifications on smartphones. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct*, MobileHCI '15, pages 652–659, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3653-6. doi: 10.1145/2786567.2793684. URL `http://doi.acm.org/10.1145/2786567.2793684`. 3.7

[321] Robin Williams. *The non-designer's design book: Design and typographic principles for the visual novice*. Pearson Education, 2015. 9.3.1

[322] Jacob O Wobbrock, Shaun K Kane, Krzysztof Z Gajos, Susumu Harada, and Jon Froehlich. Ability-based design: Concept, principles and examples. *ACM Transactions on Accessible Computing (TACCESS)*, 3(3):1–27, 2011. 2.1.1, 7.2

[323] Euphemia Wong. User interface design guidelines: 10 rules of thumb. `https://www.interaction-design.org/literature/article/user-interface-design-guidelines-10-rules-of-thumb`, 2024. Accessed: 2024-03-25. 9.3.1

[324] Jason Wu, Gabriel Reyes, Sam C White, Xiaoyi Zhang, and Jeffrey P Bigham. When can accessibility help? an exploration of accessibility feature recommendation on mobile devices. In *Proceedings of the 18th International Web for All Conference*, pages 1–12, 2021. 3.1

[325] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. Screen parsing: Towards reverse engineering of ui models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, page 470–483, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386357. doi: 10.1145/3472749.3474763. URL `https://doi.org/10.1145/3472749.`

3474763. 2.2.2, 6.2, 6.6

[326] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. Screen parsing: Towards reverse engineering of ui models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 470–483, 2021. 2.1.2, 2.2.3, 4.4.1, 4.5.1, 5.1, 7.7.3, 8.6.1

[327] Jason Wu, Titus Barik, Xiaoyi Zhang, Colin Lea, Jeffrey Nichols, and Jeffrey P Bigham. Reflow: Automatically improving touch interactions in mobile applications through pixel-based refinements. *arXiv preprint arXiv:2207.07712*, 2022. 2.2.3, 7.1

[328] Jason Wu, Rebecca Krosnick, Eldon Schoop, Amanda Swearngin, Jeffrey P Bigham, and Jeffrey Nichols. Never-ending learning of user interfaces. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–13, 2023. 2.2.3, 6.1, 8.5.3, 8.6.1, 9.7.3

[329] Jason Wu, Siyan Wang, Siman Shen, Yi-Hao Peng, Jeffrey Nichols, and Jeffrey P Bigham. Webui: A dataset for enhancing visual ui understanding with web semantics. *arXiv preprint arXiv:2301.13280*, 2023. 6.4, 6.6

[330] Jason Wu, Siyan Wang, Siman Shen, Yi-Hao Peng, Jeffrey Nichols, and Jeffrey P Bigham. Webui: A dataset for enhancing visual ui understanding with web semantics. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2023. 4.1, 8.3.1, 9.3.1, 9.3.1

[331] Jason Wu, Yi-Hao Peng, Amanda Li, Amanda Swearngin, Jeffrey P Bigham, and Jeffrey Nichols. Uiclip: A data-driven model for assessing user interface design. *arXiv preprint arXiv:2404.12500*, 2024. 9.1

[332] Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey P Bigham, and Jeffrey Nichols. Uicoder: Finetuning large language models to generate user interface code through automated feedback. In *Proceedings of 2024 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 1–14, 2024. 8.1

[333] Ziming Wu, Yulun Jiang, Yiding Liu, and Xiaojuan Ma. Predicting and diagnosing user engagement with mobile ui animation via a data-driven approach. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367080. doi: 10.1145/3313831.3376324. URL https://doi.org/10.1145/3313831.3376324. 6.2

[334] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. Uied: a hybrid tool for gui element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1655–1659, 2020. 6.4

[335] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V Le. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10687–10698, 2020. 4.2, 4.4.2

[336] Hu Xu, Saining Xie, Xiaoqing Ellen Tan, Po-Yao Huang, Russell Howes, Vasu Sharma,

Shang-Wen Li, Gargi Ghosh, Luke Zettlemoyer, and Christoph Feichtenhofer. Demystifying clip data. *arXiv preprint arXiv:2309.16671*, 2023. 9.5.1

[337] I Zeki Yalniz, Hervé Jégou, Kan Chen, Manohar Paluri, and Dhruv Mahajan. Billion-scale semi-supervised learning for image classification. *arXiv preprint arXiv:1905.00546*, 2019. 4.2, 4.4.2

[338] Jianwei Yang, Jiasen Lu, Stefan Lee, Dhruv Batra, and Devi Parikh. Graph r-cnn for scene graph generation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 670–685, 2018. 2.2.2

[339] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: Using gui screenshots for search and automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, page 183–192, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587455. doi: 10.1145/1622176.1622213. URL `https://doi.org/10.1145/1622176.1622213`. 2.1.2

[340] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192, 2009. 2.2.2

[341] Arianna Yuan and Yang Li. Modeling human visual search performance on realistic webpages using analytical and deep learning methods. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pages 1–12, 2020. 2.2.2

[342] Xiaoxue Zang, Ying Xu, and Jindong Chen. Multimodal icon annotation for mobile applications. In *Proceedings of the 23rd International Conference on Mobile Human-Computer Interaction*, pages 1–11, 2021. 2.2.2

[343] Rowan Zellers, Mark Yatskar, Sam Thomson, and Yejin Choi. Neural motifs: Scene graph parsing with global context. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5831–5840, 2018. 2.2.2, 5.3.2

[344] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12104–12113, 2022. 9.5.1

[345] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O Wobbrock. Interaction proxies for runtime repair and enhancement of mobile application accessibility. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 6024–6037, 2017. 2.1.2, 7.3, 7.4.4, 7.4.4, 8.1

[346] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021. 6.2, 6.3.2, 6.4.1, 7.1

[347] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021*

CHI Conference on Human Factors in Computing Systems, pages 1–15, 2021. 2.2.2, 2.2.2, 4.2, 4.3.2, 4.3.2, 4.4.1, 4.4.1, 4.8, 5.2, 5.5.1, 5.6, 5.7.2, 7.4.2, 7.4.3, 8.3.1, 8.6.1

[348] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. 8.5.2

[349] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024. 2.2.3

[350] Sharon Zhou, Mitchell Gordon, Ranjay Krishna, Austin Narcomey, Li F Fei-Fei, and Michael Bernstein. Hype: A benchmark for human eye perceptual evaluation of generative models. *Advances in neural information processing systems*, 32, 2019. 2.2.3

[351] Xin Zhou and Yang Li. Large-scale modeling of mobile user click behaviors using deep learning. In *Proceedings of the 15th ACM Conference on Recommender Systems*, pages 473–483, 2021. 2.2.2

[352] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points, 2019. 6.3.2

[353] Song-Chun Zhu and David Mumford. *A stochastic grammar of images*. Now Publishers Inc, 2007. 2.2.2