Understanding Screen Relationships from Screenshots of Smartphone Applications

Shirin Feiz*
Stony Brook University
Stony Brook, NY, USA
sfeizdisfani@cs.stonybrook.edu

Jason Wu†
HCI Institute, Carnegie Mellon University
Pittsburgh, PA, USA
jsonwu@cmu.edu

Xiaoyi Zhang
Apple
Cupertino, CA, USA
xiaoyiz@apple.com

Amanda Swearngin
Apple
Cupertino, CA, USA
amaswea@cs.washington.edu

Titus Barik
Apple
Cupertino, CA, USA
titus.barik@apple.com

Jeffrey Nichols
Apple
Cupertino, CA, USA
jwnichols@apple.com

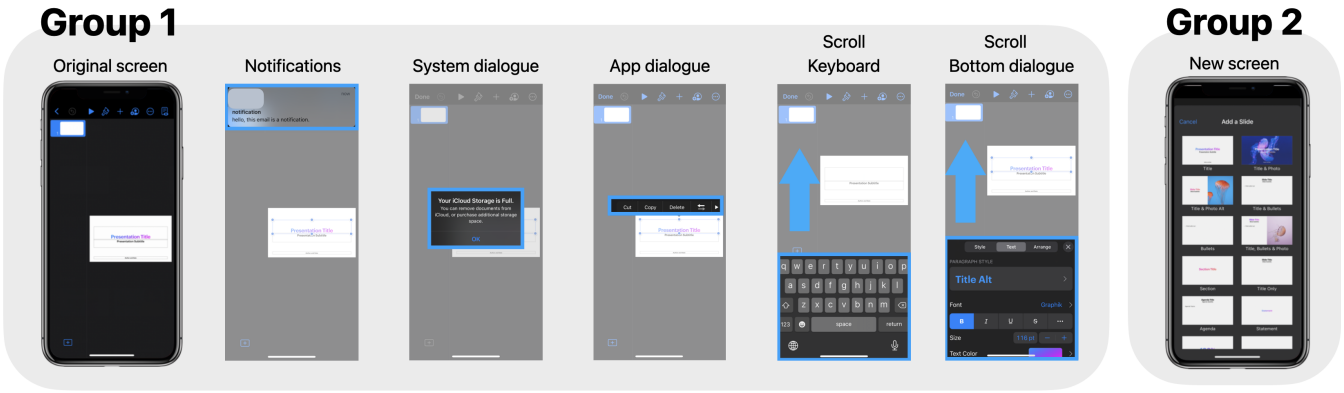

Figure 1: Screenshots of the keynote app separated into two groups of "same" screens. Screenshots in Group 1 demonstrate six types of events on the "original screen". In this paper we present two models: (1) a screen similarity model to recognize instances of the same screen from a collection of screenshots from a single app, and (2) a screen transition model to identify different types of events that appear in an interaction trace.

ABSTRACT

All graphical user interfaces are comprised of one or more screens that may be shown to the user depending on their interactions. Identifying different screens of an app and understanding the type of changes that happen on the screens is a challenging task that can be applied in many areas including automatic app crawling, playback of app automation macros and large scale app dataset analysis. For example, an automated app crawler needs to understand if the screen it is currently viewing is the same as any previous screen that it has encountered, so it can focus its efforts on portions of the app that it has not yet explored. Moreover, identifying the type of change on the screen, such as whether any dialogues or keyboards have opened or closed, is useful for an automatic crawler to handle such events while crawling. Understanding screen relationships is a difficult task as instances of the same screen may have visual and structural variation, for example due to different content in a database-backed application, scrolling, dialog boxes opening or closing, or content loading delays. At the same time, instances of different screens from the same app may share some similarities in terms of design, structure, and content. This paper uses a dataset of screenshots from more than 1K iPhone applications to train two ML models that understand similarity in different ways: (1) a screen similarity model that combines a UI object detector with a transformer model architecture to recognize instances of the same screen from a collection of screenshots from a single app, and (2) a screen transition model that uses a siamese network architecture to identify both similarity and three types of events that appear in an interaction trace: the keyboard or a dialog box appearing or disappearing, and scrolling. Our models achieve an F1 score of 0.83 on the screen similarity task, improving on comparable baselines, and an average F1 score of 0.71 across all events in the transition task.

*This work was done while Shirin Feiz was an intern at Apple.
†This work was done while Jason Wu was an intern at Apple.

IUI '22, March 21–25, 2022, Helsinki, Finland

ACM ISBN 978-1-4503-9144-3/22/03.
https://doi.org/10.1145/3490099.3511109

CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**.

KEYWORDS

user interface similarity, ui modeling, ui semantics

ACM Reference Format:
Shirin Feiz, Jason Wu, Xiaoyi Zhang, Amanda Swearngin, Titus Barik, and Jeffrey Nichols. 2022. Understanding Screen Relationships from Screenshots of Smartphone Applications. In *27th International Conference on Intelligent User Interfaces (IUI '22), March 21–25, 2022, Helsinki, Finland*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3490099.3511109

1 INTRODUCTION

Systems that seek to understand an app user interface must be able to identify the different screens of an app and understand how those different screens are related to each other. Understanding an app user interface in this way is useful in a number of different settings. For example, an automated app crawler needs to understand if the screen it is currently viewing is the same as any previous screen that it has encountered, so it can focus its efforts on portions of the app that it has not yet explored. Similarly, systems that replay previously recorded UI actions must be able to identify whether a screen seen in the recording is the same as the screen seen during playback in order to know whether the replay is proceeding successfully. Finally, UI analysis tools can be improved with techniques that employ screen detection to reduce noise, which could be particular useful for gaining insights from large UI datasets, such as Rico [9, 10].

In our work, we focus on determining screen relationships solely from pixel information, specifically through screenshots. We choose this approach because it can be applied generally across many different types of apps regardless of their underlying implementation and choice of UI toolkits. Other methods have made use of underlying representations, such as view hierarchies on the Android platform. For instance, [9, 22, 26, 39] use view hierarchy information to determine the available UI elements and their layout on the screen. Unfortunately, this method is specific to Android user interfaces that expose a well-formed view hierarchy, such as those built with the standard toolkit, and may not be applicable to UIs implemented in alternate toolkits, such as Flutter.

Detecting whether two screens are similar is a challenging problem, and can be defined in different ways for different tasks. For example, in a design search task as in Rico [9], Swire [18] or Screen2Vec [21], the goal is to search a dataset of many apps using a query screenshot to find many screens with similar elements and functionality from many different kinds of apps. In our work, we focus on the screen similarity task within apps, which is relevant for systems that must automatically navigate within an app. These systems must be able to identify whether a screen they have visited before is the same as the screen they are currently visiting, even if the content of that screen differs due to changes in the UI state. For example, an instance of a shopping cart screen in a retail app must be recognizable as the same screen regardless of how many items appear in the cart. Similarly, instances of a product screen should be recognizable as the same screen if the same layout template is used, even if the product being displayed is different. Different scroll positions of the screen or the presence of a dialog box or an on-screen keyboard can further complicate this problem.

In this paper, we train two machine learning models to infer whether two screens are similar and to identify whether any of 6 common events have occurred. We formulate these two problems as follows:

- Screen Similarity: A binary classification problem in which two screenshots are given as input and the task is to predict whether they are images of the same screen or not. The only assumption is that the screenshots are from the same app.
- Screen Transition: Assumes two input images are consecutive screenshots collected during an interaction trace. The binary classification of same screen or not remains a part of the problem, but we introduce the additional sub-problem of recognizing whether one or more of six common events have occurred including scrolling, and the appearance or disappearance of app dialog boxes, system dialog boxes, bottom dialog boxes, notifications, keyboards. We chose these six events based on our own inspection of sequences of interaction data from 1,110 apps. This extra information is useful to help guide automation and crawling algorithms as they interact with an app.

In order to train our models, we collected a dataset of over 77k screenshots from 1,110 different iPhone applications, which were produced by human crowd workers traversing each app for approximately 15 minutes. A separate group of crowd workers annotated this data with similarity and transition information.

Our screen similarity model is a combination of a UI element detector [28], which extracts and featurizes on-screen UI elements from a screenshot, and a transformer [31] that jointly encodes and classifies the relationship between two screens. Our approach is informed by prior work in UI understanding [34] and text-based visual question answering [17, 29, 30], both of which model the relationship between several multi-modal elements. Our results show that our model achieved an F1 score of 0.83 over our datset, outperforming existing methods.

Our screen transition model is a siamese network [13] that is trained to generate an embedding vector from an app UI's screenshot. The goal is for the difference between two screen embeddings to encode the presence of one of the six event types, and thus predict the transitions between two related screens (*e.g.,* the appearance of an application or system dialog). We train this model on both the screen similarity and transition problems. While we find that its performance on screen similarity lags the transformer model described above, we also found that the inclusion of this problem during training resulted in better model performance on the transition task than when the model was trained on the transition task alone. We also found that our set of six events were difficult for our model to distinguish, and combined the dialog and notification events into a single category resulting in three final types. To the best of our knowledge, our screen transition model is first of its kind. It achieves an average F1 score of 0.71 across the 3 transition types.

This paper begins with an examination of related work, which provides inspiration and also the baselines that we use to evaluate

the screen similarity model. We then discuss our data collection before diving into the screen similarity model in depth, describing the annotation process, baselines, model training and evaluation. We then discuss our screen transition model from annotation through evaluation, our results as a whole, and then conclude.

2 RELATED WORK

The vast majority of systems have not used pixels to determine whether screens were similar, but instead resorted to using available underlying representations of the user interface, such as view hierarchies on the Android platform. Our work relies entirely on the pixels of the user interface, allowing it to operate when a non-standard UI toolkit is used to build the user interface or if the underlying data is not available due to platform limitations.

This previous research has defined "similar screens" in various ways and has presented methods for identifying similar screens as a sub-goal across several different applications. These applications have included user interface testing, bug report understanding, and user interface dataset search.

2.1 User Interface Testing & Bug Reports

Automatic UI testing often has a need for identifying the relationship between the screens of the app to understand the structure of the app. MobiGuitar [2], an early Android-based automated test tool, made use of "GUI Ripping" to extract the state machine of an app. This tool identifies similar states as they are encountered, and ultimately produces a directed graph of UI states. Determining whether a state is the same included understanding the screen, which was accomplished by inspecting the view hierarchy and other underlying data about the interface specific to the Android platform. Other crawlers [6, 19, 23], including DroidBot [22], and other crawling algorithms [33, 36], including NFS [25], also rely on similar methods of using the view hierarchy to identify that an app is the same state. For instance, PUMA [14] and DECAF [24] define an encoding of the structure of the view hierarchy and define similarity based on the distance between the embeddings. Our work seeks to achieve a similar goal, but uses only pixel information and not any additional underlying UI data. These systems are also frequently evaluated in terms of code coverage or other test-related metrics, which makes it difficult to determine how successfully their similar state algorithms function and also complicates direct comparison with our work.

Another UI testing system called AppFlow [16] synthesizes reusable UI tests across different apps in a similar category, such as shopping cart apps. AppFlow includes a screen recognition model that uses screen layout represented as a string, OCR, and Android Activity titles to train a model to recognize similar screen types across applications. Pixel information is only used via OCR to extract text from the user interface, and it also relies on underlying UI representations that are only available from the Android platform.

Several systems have sought to help developers manage bug reports in various ways through joint modeling of screenshot features and textual bug reports. Yu et al. [37] help developers prioritize bug reports, similarly ReCDroid [40] automatically recreates crashes. Work by Wang et al. [32] uses a multi-modal fusion of screenshot features and textual bug reports to identify duplicate bug reports. In this work, they do not feed images directly into the model, but instead extract two image features and use them as inputs. They do compare image-only and text-only approaches to their fused result and find that the image-only detector generally underperformed the text-based classifier. All of these systems rely on bug report text as an important input to their process, whereas our work relies solely only on image data.

2.2 UI Understanding

A number of systems have explored helping developers and designers understand existing app user interfaces, either at the overall application level or specifically at the screen level. The motivation for this work is often to provide inspiration for a new design through exploration of existing potentially related designs. For instance, [5] creates a searchable gallery of UI element designs based on the app screenshots along with information regarding each application. Other works has been inspired by the availability of large-scale datasets, such as Rico [9], which are too large to browse effectively.

From App Metadata & View Hierarchies:

Many systems in this area have relied on app metadata or Android view hierarchies as the basis for their operation.

StoryDroid [7] extracts a storyboard of an Android app from the structured data included in an app's APK file. The extracted storyboard can then be displayed as a reference for app designers or used to compare one or more apps with one another. Our work in this paper can be reapplied to extract similar storyboards, though entirely from screenshots and without requiring inspection of the app binary. While StoryDroid focuses on the design of an entire app, much more work has explored search in terms of a single screen. Like the systems we've seen already, many rely on underlying UI data to learn a searchable representation of a screen at various levels of complexity.

NEAR [35] seeks to detect near-duplicate pages on the web. It addresses much the same problem as we examine in this paper, except that it uses the DOM tree as its primary input and works only in the web context.

Zhang et al. [39] define a set of screen equivalency heuristics based on identifiers and semantic structures found in Android view hierarchies to determine screen equivalence for accessibility annotation.

The creators of Rico [9] used the Android view hierarchy to compute a basic image representation of screens from the UI's textual and non-textual elements, and trained an autoencoder based on this representation. The autoencoder was shown to be useful for UI screen search across the dataset.

Building on this work, Liu et al. [26] learned deeper semantic models of UI elements from the view hierarchy and icons from pixels. They used these in combination to produce a semantic representation of a screen, which their model fed into a autoencoder similar to the Rico approach. They demonstrated this produced superior results to the simpler Rico model.

Enrico [20] extends this work even further and explores topic modeling of screens, also using an autoencoder-based approach.

Topics cross different apps and are useful for a number of applications, but do not seem to be at a fine-grained enough level to be applied to the screen similarity problem discussed here.

Screen2Vec [21] builds on these autoencoder-based approaches, adds multi-modal features from both the textual and layout content of the UI in a two-layer model, and is trained via a self-supervised method. Screen similarity for the purpose of UI search across a dataset is explored, among other applications. Screen2Vec, like the other methods described so far, relies entirely on underlying view hierarchy information available on the Android platform and does not use any pixel data in its inputs.

From Pixels:

Some systems have explored understanding the user interface from pixels alone, but have limitations for our application area.

PuppetDroid [12] uses a perceptual hash to identify similar screens, courtesy of the pHash library [1]. Hashes were used to identify similar screens of known good apps in suspected malware apps as part of a method to detect malware. We include pHash, as well as other hashing methods from a common image hash library (wHash and dHash), as baseline methods to evaluate the performance of our screen similarity model.

Chiatti et al. [8] explores clustering of screenshots across apps using unsupervised and semi-supervised methods. Their primary focus is to create clusters that would assist in a behavioral usage study, and it does not seem that their clusters would have enough granularity to be applicable to our screen similarity problem.

Malisa et al. [27] attempt to determine whether a login screen is spoofed or not based on an understanding of visual cues that humans tend to ignore. They train a model based on features extracted from screenshots that are specifically chosen to identify visual cues that are likely to be ignored by human viewers. This work is related to our method in that it compares a reference screen to the currently displayed screen, but the assumptions are very application-specific.

Screen Recognition [38] is a system that detects the location and type of UI elements on an iOS screen entirely from the on-screen pixels. The result is used in accessibility services when the current app does not contain any accessibility meta-data. This work does not attempt to detect similar screens directly, however its output could be used as a source of data similar to the view hierarchies used in the systems described earlier. One of our baseline similarity algorithms uses a similar object detection algorithm and then heuristically compares the elements that are identified across two screens.

Screen Parsing [34] is another system in a similar vein, which detects UI elements from pixels and also infers their hierarchy. Similarly VINS [3] applies object detection to detect UI elements and a neural network to learn the layout structure. UI similarity search are demonstrated applications of these works, though the focus seems to be on search across screens from different apps rather than within a single app. In addition, both of these methods focus on characterizing screens by their layout information (*i.e.,* location and type of on-screen elements) and do not take into account additional visual information such as element appearance (*e.g.,* element color) that could provide additional cues for our task. In contrast, the featurized vectors produced by our UI object detector do incorporate appearance, which are then consumed by the transformer portion of our screen similarity model.

Swire [18] enables searching a dataset of GUI screenshots using a sketch image as input. This is accomplished by training a joint embedding between sketch images and real screenshots using a triplet loss function. Swire's method is similar to the siamese network that we use for our screen transition model, however the focus again is on searching across screens from different apps rather than within a single app.

3 DATA COLLECTION

The same app dataset is used as the basis for all of our work in this paper, so we describe its collection first before separately examining how we further annotate and train models for the similarity and transition problems.

We collected a dataset of screenshots from 1,110 popular free iPhone apps as listed in the Apple App Store. We excluded apps from the various Game and AR app categories, as these apps often have non-traditional interfaces without clearly segmented screens.

A third-party data collection company collected the dataset, which our organization paid a large flat-fee for the entire job. That company in turn employed a set of 36 English speaking crowd workers from the United States, who were paid at least the minimum wage for their locale (exact payment varied based on the volume of data collected).

Workers visited a web site, through which multiple tasks were available. Each task presented a real iPhone device with which the workers could interact using a remote desktop-style interface. For each task, we asked them to explore the presented app as thoroughly as possible within 15 minutes. Our custom data collection tool captured automatic screenshots every second provided the screen content had changed.

We post-processed the data set and dropped any duplicate consecutive screenshots. To do so, we examined the pixel similarity of each pair of two consecutive screenshots. First, we discarded the top status bar of both screenshots, because this area often contains information that may change but is unrelated to the app, such as the time and battery level. We then applied a template matching algorithm with normalized correlation coefficient to compare the images and used an empirical threshold (98%) to detect and drop duplicates. The final dataset is comprised of 77,655 non-identical screenshots.

4 SCREEN SIMILARITY

To find whether two screens of an application are the same, we (1) carried out an annotation task to determine which screens in our collected dataset were the same, (2) used the resulting annotated dataset to train a model that can predict screen similarity, and (3) evaluated our model in comparison to previous methods.

4.1 Screen Similarity: Annotation

We annotated screenshots of each app into groups of same screens. To make the annotation task easier, workers first annotated each pair of consecutive screenshots as either same or new in order to identify and merge sequences of consecutive same screenshots in

[1] https://www.phash.org

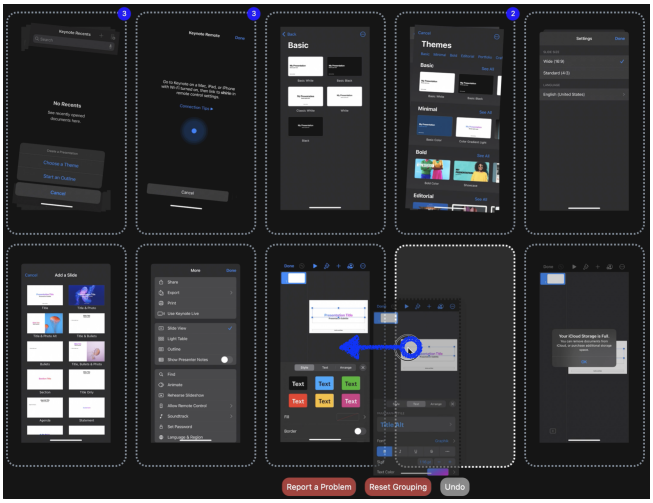

Figure 2: Annotation interface for grouping same screenshots of an app as a card-sorting task. The annotator can drag and drop screenshots (cards) to build stacks of same screenshots.

the user interaction trace. In this step, the workers marked 49% of the screenshot pairs the same.

We built an annotation interface to enable workers to group screenshots as a card sorting task (see Figure 2). This interface allows the annotator to group same screenshots with drag and drop actions. Splits and merges of existing groups are supported in this interface. Also, annotators can expand current groups to re-visit the screenshots or move any mis-grouped screenshots and place them in another group.

The workers used our interface to group the screenshots of a single app in the dataset, and we instructed them to stop when each group of screenshots in their opinion represented a different screen. We gave the workers documentation and training prior to the actual tasks to ensure that they were familiar with our definition of a same screen. There were a total of 1,110 tasks (one task per app) with an average of 32.19 screens in each task.

We divided the set of apps into 10 batches for annotation. After the workers annotated all the data, our Quality Assurance (QA) team went over a random 10% subset of the annotations for each batch and evaluated the annotated groups. They used two metrics to evaluate the groupings: *within-group accuracy* and *between-group accuracy*. For within-group accuracy, we presented the QA team with annotated groups and asked to identify if any screenshots were incorrectly added to a group. For between-group accuracy, the QA team evaluated if separate groups are in fact representing different screens. To do so, we presented them with the final annotated groups of an app and asked if any groups should be merged together. The QA evaluation for the annotated groups shows that the groups have high within group accuracy (over 98%) which suggests that most screens that are grouped together were marked as correctly representing same screens. However, the QA evaluation for the between group accuracy is at 34.32%, which suggests that the QA team would merge the current annotated groups further than the original annotators.

To create a final set of grouping annotations, we consider two screenshots to belong to the same group if both annotators put the pair in the same group. We chose to drop all screenshot pairs where annotators disagreed from our dataset, because of the low between-group accuracy observed in the annotations and the high likelihood that such examples might introduce substantial noise. The final set of consensus groups has an average of 6.94 groups per app (standard deviation: 2.63) with an average of 3.09 screenshots per group (standard deviation: 1.15).

Finally, in order maximize the size of our similarity dataset, we (1) added all consecutive "same" screenshots to our grouped screens, and (2) created a dataset containing every combination of two screens from the annotated groups and labeled them as same if they belonged to the same group and different otherwise. Going forward, we refer to this dataset as the **screen similarity dataset**.

4.2 Screen Similarity: Baseline Methods

Before training our own models, we explored how several existing methods performed on our dataset. We chose these methods since they were fully pixel-based and had been used in previous work to solve problems very close to the similarity problem we wish to solve here. These methods are also reflective of standard methods available in the field to gauge similarity between images. Image hashing is a classic method for testing image similarity, which has several common implementations. The UI element detection method builds on recent work in UI object detection [38] and is based on previous work within our team that is currently used in a production use case. Finally, TensorFlow Similarity is a state-of-the-art library for training deep ML models on a variety of image similarity problems. We also attempted to build a similarity model building on both recent advances in UI object detection and the autoencoder-based methods initially pioneered by Rico [9], however we were not successful, possibly due the relatively smaller size of our dataset compared to those previously used.

Image Hash: We used the Image Hash python library [1] which is designed to generate similar hashes (fingerprints) for 'similar' input even in presence of minor pixel value changes. Specifically, we used the following hashing methods to determine similarity of screenshots in the screen groups dataset: Perceptual Hash (P-Hash), Wavelet Hash (W-Hash), and Difference Hash (D-Hash). We ran all methods with their default settings.

UI Element Matching: This is a heuristic-based method to determine screen similarity based on the bounding boxes of UI elements as detected by UI object detection model. This method was originally created by our team for use in app crawling applications and has been used for over a year in our systems, but we have never formally evaluated it. For this paper, we start by extracting the UI elements for all screenshots in our screen similarity dataset using a pre-trained model based on the implementation of Screen Recognition [38]. To determine whether two screens are similar, the algorithm examines the detected UI elements from the two screens, and tries to find matches between the them in terms of type and bounding box. UI Elements are matched based on their type (*e.g.*, both are text fields) and bounding box position, and if

almost all of the elements between two screens have good matches, we declare them the same. Based on manual inspection on a small set of examples, we set some parameters for our heuristic approach: we used an IoU threshold of 0.9 to determine bounding box overlap, and we required all but two UI elements to be matched in order for two screens to be the same.

Tensorflow Similarity: Tensorflow Similarity is a metric learning library that is designed specifically to learn models of similarity between different groups of images [4]. We used this library to train a model from our app screen dataset that learned a distance function between two screens, which is used to determine similarity. We set our model hyperparameters (embedding size of 64) based on example code and documentation released with the dataset.

Autoencoder-Based Similarity: We attempted to replicate the Rico autoencoder design used by previous systems [9, 20, 21], but without relying on view hierarchies as those systems did. In the place of view hierarchies, we use the UI objects in each screenshot as detected using the pre-trained model based on Screen Recognition [38] that was also used in the UI Element Matching baseline above. We first attempted to train an autoencoder using images constructed from the detected UI elements with the plan to train an binary classifier using the resulting pre-trained encoder duplicated twice as inputs. Unfortunately, we were not able to successfully train a reliable autoencoder using the Rico design from our dataset, possibly because of the large linear design of the Rico autoencoder and our relatively smaller dataset (1,110 apps in our dataset vs. around 10,000 in Rico). We may attempt this method in the future if we increase the size of our dataset.

The performance of these methods is discussed later in more depth, however none of these methods were able to achieve an F1 score greater than 0.7 on our test set split, with the majority performing much worse. As a result we chose to pursue developing our own method.

4.3 Screen Similarity: Modeling

Our system for determining screen similarity uses two models: (1) a UI element detector that extracts and featurizes on-screen UI elements from a screenshot, and (2) a transformer that jointly encodes and classifies the relationship between two screens. Our approach is informed by prior work in UI understanding [34] and text-based visual question answering [17, 29, 30], both of which model the relationship between several multi-modal elements.

UI Element Detection: We first use an object detection model to locate UI elements from a screenshot, which allows our system to attend to locations of the input image that are most likely to contain relevant information. We use a Faster-RCNN model [28] which was pre-trained on a large dataset (~77k annotated screenshots) of popular iOS app screens [38]. Note that the dataset used to train the element detection model is completely separate from our screen similarity dataset. We found that this model worked well "out of the box" for the screens in our grouping dataset without any modification or finetuning. We first resized each screenshot to a fixed size (256x256) and normalized it before feeding it into the model. We post-processed the model output by filtering out objects with a low confidence score (less than 0.7) and applied non-max suppression to remove overlapping detections (IoU greater than 0.2).

For each of the detected UI elements, we extract its (1) position, (2) predicted label, and (3) a fixed-size feature vector extracted from the model's activations (we used the output of the `fc6` layer, following previous work [29]).

Similarity Transformer: To determine the relationship between two screens (*i.e.,* whether they are the same or different screens), we extract their UI elements and jointly encode them with a transformer model. The transformer model uses attention [31] to contextually encode UI elements and learn relationships between elements of both screens.

Model Architecture: Our similarity model consists of several attribute-specific encoders (implemented as linear layers) and a transformer encoder, see Figure 3. Our model computes several embeddings for each UI element based on its: (1) position, (2) class label, (3) RCNN feature vector (encodes visual appearance), and (4) screen (*i.e.* which of the screen it belongs to). The model then combines these embeddings to produce a fixed-size representation for each UI element. Similar to other transformer-based models, a "class token" (`[CLS]`) is inserted, which the model encodes along with the rest of the input to enable the model to make predictions. Finally, the model feeds the final embedding of `[CLS]` into a linear classifier which predicts if the two input screens are the same screen or not.

Training Procedure: We trained the Similarity Transformer model on our screen similarity dataset (Section 4.1). We used our UI Element detector to pre-compute the detections and features for each screenshot. Each example consisted of two featurized screens and a label describing their relationship. We trained our model to predict the similarity label from the two input screens by minimizing the binary cross-entropy loss.

$$\mathcal{L}_{sim} = \text{BCE}(\hat{y}_{sim}, y_{sim}) \tag{1}$$

where y_{sim} and $\hat{y}_{sim}$ are the similarity label and predicted similarity label respectively. In this equation, BCE refers to the binary cross entropy loss function (commonly used for multi-label classification), which we use to learn the probability that the two screens are the same.

In addition, we added a secondary "masked prediction objective" [11, 30] to further improve the performance and generalizability of our model. During training, a subset of the model's input is randomly "masked" with a probability of 15% [11] by replacing its attribute values with 0. The model is trained to recover the masked elements using contextual information. Specifically, the model predicts the *(i)* position, *(ii)* class label, and *(iii)* RCNN features of masked elements.

$$\begin{aligned}\mathcal{L}_{masked} \quad = \quad & \text{MSE}(\hat{y}_{pos}, y_{pos}) + \text{CE}(\hat{y}_{label}, y_{label}) \\ & + \text{MSE}(\hat{y}_{RCNN}, y_{RCNN})\end{aligned} \tag{2}$$

MSE is the Mean Square Error, *CE* is the Cross Entropy. The final loss function is the sum of the similarity and masked prediction objectives.

$$\mathcal{L} = \mathcal{L}_{sim} + \mathcal{L}_{masked} \tag{3}$$

We trained our model with a learning rate of 5e-5, following the learning rate suggestion from the original BERT paper [11],

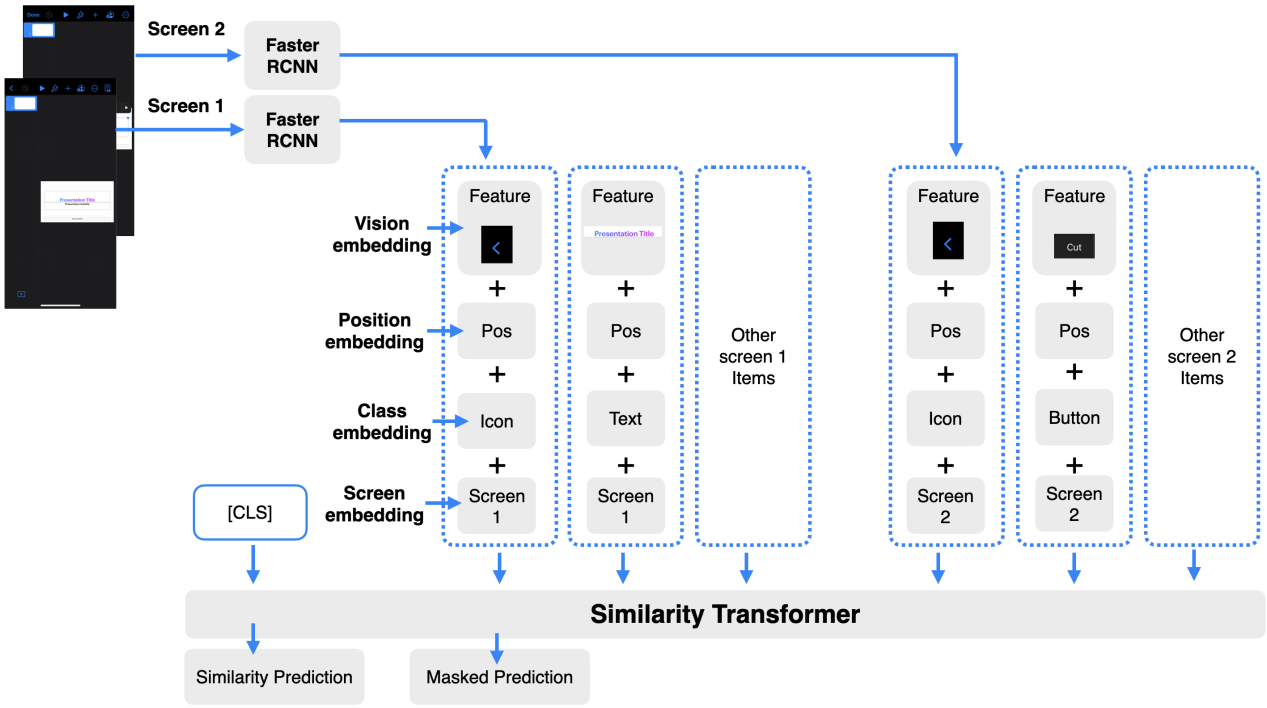

Figure 3: Similarity Transformer model architecture. We use a transformer model to jointly encode the UI elements of two screens. UI elements are extracted and featurized from screenshots using a Faster-RCNN model. UI elements are encoded using several embeddings (vision, position, class, and screen) which captures information from different modalities. A class token (`[CLS]`) is prepended to the sequence, which is used to predict the relationship between the two input screens. To improve performance and generalization, we incorporate a masked prediction objective which requires the model to predict the attributes (*e.g.*, position, class) of a randomly selected subset of elements which have been replaced with 0.

Table 1: Comparison of different similarity detection techniques with the screen similarity test set.

	Precision	Recall	F1
P-hash	0.93	0.14	0.24
W-hash	0.92	0.20	0.33
D-hash	0.95	0.13	0.24
UI Element Matching	0.92	0.17	0.29
Tensorflow Similarity	0.71	0.68	0.69
Similarity Transformer	**0.80**	**0.86**	**0.83**

with a hidden size of 256 which was found by automated search. Following transformer-style papers the AdamW optimizer was used. The number of epochs (78) was determined by early stopping, where training stops when validation F1 score did not improve for 5 epochs.

Our model was able to distinguish same screens with an F1 score of 0.83 for the screenshot pairs.

4.4 Screen Similarity: Evaluation

We compared our screen similarity model with the baselines described earlier. Table 1 shows the results for determining screen similarity. We split our screen similarity dataset into training (70%), validation (15%), and test (15%) sets. We partitioned our dataset so that all screens from any one app appear in the same split. We use the training split to train all models, including both the Tensorflow Similarity and Screen Similarity models. We evaluate all methods with the test set partition of the dataset. Our results show that our Screen Similarity model outperforms all other baseline methods with an F1 score of 0.83.

Our results show that the image hash methods and UI Element matching method all yield high precision (over 0.92) while falling short on recall (between 0.13 to 0.17). This suggests that these methods will miss a large portion of same screens. This is likely because these methods are sensitive to the visual appearance of the screens, whereas some same screens may have content differences that substantially change their visual appearance. The Tensorflow Similarity methods are less sensitive to appearance and utilize deep neural networks, which could suggest that they would have better generalization power and be less dependent on the pixel similarity of the screenshots. The comparison between the Screen Similarity model and Tensorflow Similarity shows that screen similarity has

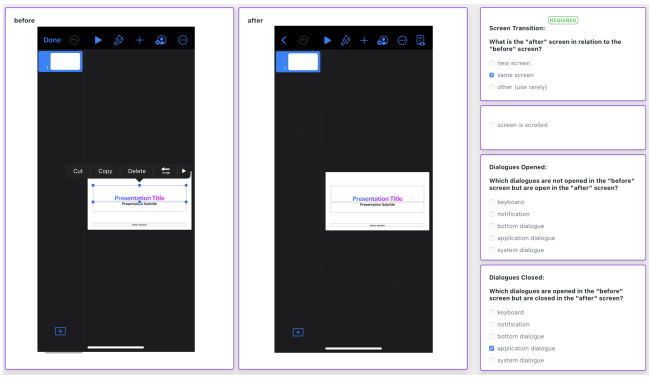

Figure 4: Interface for annotating sequences of screenshots. The annotator identifies if the screenshots are 'same' or 'new' or 'other'. For 'same' screens, the annotator marks whether the screen is 'scrolled'. The annotator marks all dialogue opening and closing that is applicable to the pair.

much higher precision and recall increasing the F1 score from 0.69 to 0.86.

The overall comparison of the models shows that the proposed Screen Similarity method has the highest F1 score of 0.83 over the screen similarity dataset.

5 SCREEN TRANSITION

In this section, we focus on the relationship between consecutive screenshots in a user interaction trace which is motivated by applications such as automated app crawling or automatic replay of UI actions. In particular, we focus on screen relationship in terms of "screen transition" in which we introduce the additional sub-problem of recognizing events such as scrolling, dialogues opening or closing. This extra information is useful to help guide automation and crawling algorithms as they interact with an app.

Transition types: We came upon this sub-problem while examining the data collected from an initial set of user interaction traces, when we noticed that the app UI sometimes changed in substantial ways even when the underlying screen did not. We further examined the dataset, identified events that occurred, and discussed these events amongst ourselves until we agreed upon a set of 6 types that seemed distinct and common amongst the data that we examined. We identified 6 types of events: notification, system dialog, app dialog, keyboard, bottom dialog, and scrolling (see Figure 1). These events in most cases seem to be independent of screen similarity, as a dialog box or the keyboard might appear regardless of whether the app transitions to a new screen or remains on the same screen. Only the scrolling event requires the app to remain on the same screen when the event occurs.

In this section, we (1) annotate consecutive sequences of screenshots from a user interaction trace, (2) train a model to detect the transition type between two consecutive screenshots, and (3) evaluate our model. We also compare the task of same screen detection between randomly selected pairs of screenshots versus screenshots that come from a sequence.

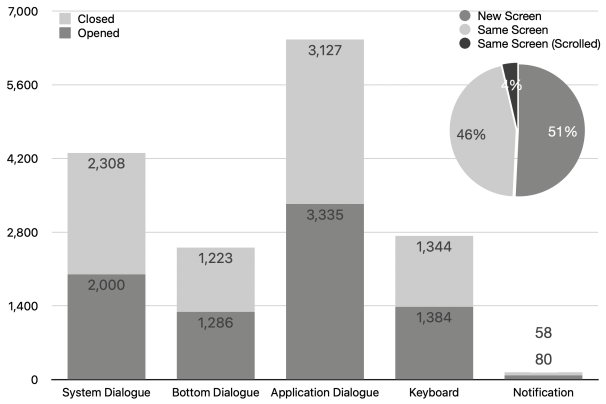

Figure 5: Screen transition dataset annotation totals, including opened and closed, for each of the five transition types. Application dialogues were most common type of transition in our dataset and notifications were the least common type of transition in our dataset. The number of Opened and Closed examples were relatively balanced for each type of transition. Our dataset contained slightly more examples of new-screen pairs than same-screen pairs.

5.1 Screen Transition: Annotation

We use the same collection of screenshots described in section 4.1 to create our screen transition dataset as follows. In this task, our annotators annotated the 1,110 sequences of screenshots, where each sequence captures user interactions within one app. We focused on two components for annotation of screenshot sequences: (1) if two consecutive screenshot are the same, and, if so, (2) whether the screen is scrolled in case the screens are the same, or (3) any dialog boxes or the keyboard has opened or closed. In the case of finding the same screen, it is important to know if there has been any scrolling between the screens since it is critical in terms of matching items of two screens in many applications. We also had our annotators label the dialogue transitions because they are common types of changes between screens and recognizing if a dialogue has opened or closed can facilitate automatic interactions with screens in applications, such as closing an errant dialog box. Figure 1 shows different types of transitions. Note that more than one transition might happen between two consecutive screenshots.

For each sequence, we consider every pair of consecutive screenshots and refer to the screens in each pair as $screen_{before}$ and $screen_{after}$ to indicate their order. For each pair, the $screen_{after}$ is marked as either same, new or other in relation to the $screen_{before}$. The definitions for same versus new screen is the same as the task described at section 4.1. The same label indicates that both the screenshots are showing the same screen although some differences might exist. The new label is used whenever the $screen_{after}$ shows a new screen compared to the $screen_{before}$. The other label is rarely used in case any of the screenshots are not clearly visible, such as when taken during animation or before the screen is properly rendered. In addition, for every pair of same screens, we had our annotators identify the six common types of transitions, including scrolling and the opening and closing of 5 items:

system dialogues, notifications, bottom dialogues, keyboards, and application dialogues.

Figure 4 shows the annotation interface for this task. For this annotation task we used a web-based interface where annotators can examine the before and after screens and mark the annotation. Specifically, we required the annotators to choose one of the same screen, new screen, or other options. In case the annotator selected same screen, we asked them to mark if there has been any scrolling between the two screenshots or whether any combination of dialogue opening and closing had occurred.

We divided the dataset into 20 similar-sized batches and performed the annotation process on each batch. The annotation process started with two separate workers annotating each pair of consecutive screenshots. To monitor the quality of the annotation labels, our QA team randomly examined 10% of the annotations and provided a detailed QA report for each of the 20 batches of annotations. For each batch, we shared clarifications regarding common errors with annotators to reduce error. The average annotation label accuracy across the batches was 95.33% (std: 0.58 %).

For any screenshot pair with at least one disagreement on their annotations, our QA team examined the pair and annotated the correct labels. On average the screenshot pairs with disagreement were 22.4% with standard deviation of 2.39% across the batches. Over the dataset, the disagreement rate between the 'same screen' and 'new screen' was 8%.

The annotators assigned the label 'other' to 18 screen pairs, which we dropped from the dataset. Figure 5 visualizes the distribution of labels in this dataset.

5.2 Screen Transition: Modeling

We initially used our Screen Similarity Transformer to predict screen transitions by adding a classification head to the `[CLS]` token; however, we found that many visual cues for screen transitions (*e.g.,* screen dimming) involved changes that occurred outside of UI element boundaries. Since our transformer model featurizes screens as sets of UI elements, it is unable to capture these signals. Thus, we trained a separate model to predict transitions between screens from the entire screenshot.

Using our dataset of annotated screen sequences, we trained a separate machine learning model, which we refer to as the Screen Transition model, that encodes the entire UI screenshot, instead of focusing on regions contained by UI elements. Our Screen Transition model is a *siamese* network [13] trained to produce an embedding space that reflects both screen similarity and the type of screen transition. Our network architecture consists of a CNN-based encoder (ResNet-18 model [15]) followed by a classifier (Fig 6).

We were primarily interested in the Screen Transition model's ability to classify transition type; however, we found that incorporating screen similarity labels during training improved performance.

To train the Screen Transition model, we iterated through our dataset of annotated screen pairs and computed a "difference vector" Δh for each pair by subtracting the embedding of the first screen from the second. We used Δh to characterize the similarity and transition relationship between the pair of screens.

A pairwise contrastive loss [13] was used to encourage similar screens to be close (*i.e.,* within a distance m) in embedding space.

$$\mathcal{L}_{sim} = \begin{cases} ||\Delta h||_2 & \text{if } s_1 = s_2 \\ \max(0, m - ||\Delta h||_2) & \text{otherwise} \end{cases} \quad (4)$$

Δh is also fed into a multi-layered feedforward network which predicts the presence of transitions. We jointly optimized both networks by computing a weighted sum of their loss functions.

$$\mathcal{L} = \mathcal{L}_{sim} + \lambda \cdot \text{BCE}(\hat{y}, y; w) \quad (5)$$

Because our dataset is imbalanced (*i.e.,* often contains many more negative examples than positive examples for each transition type), we computed a weight vector w based on the transition frequencies in the training data.

5.3 Screen Transition: Evaluation

We split our screen transition dataset into training (70%), validation (15%), and test (15%) sets. We partitioned our dataset so that all screens from any one app appear in the same split.

We evaluated our transition prediction model on the screen transition dataset and found that our screen transition model classifies transitions with an average F1 score of 0.65 across 6 transition types (a total of 11 classes, since dialogues and keyboards can be both opened and closed), while our transformer model achieved an F1 score of 0.46. We also considered a detection transition types in a reduced format by merging all types of dialogues together (*i.e.,* app dialogues, bottom dialogues, and system dialogues were merged into one class). We hypothesized that a smaller number of classes would be easier for the model to learn, and for some applications, these types of transitions could be treated similarly, since they could all be "dismissed." For the reduced set of classes, our screen transition model achieved an average F1 score of 0.71 across 3 transition types (5 total classes). Table 2 shows the F1 score for each of the transitions. We also achieved an F1 score of 0.80 for identifying same screens in our screen transition dataset.

For the screen transition dataset, both screen similarity and screen transition models can predict whether two consecutive screens are the same. To compare the two, we evaluated our model trained on the screen similarity dataset and tested with the the screen transition dataset to identify same screens. The model performed better than it originally did on the screen *similarity* dataset and achieved an F1 score of 0.86 which is higher than how it performed over the screen similarity dataset with F1 score of 0.83. In addition, we used the same model architecture and re-trained it (from scratch) on the screen sequence dataset. Because the screen sequence dataset is much smaller, we used a smaller batch size (32) and kept other hyperparameters constant. However, the performance dropped significantly, and this model was only able to reach an F1 score of 0.80. We believe there are two likely causes for this: *(i)* the screen sequence dataset is significantly smaller, and *(ii)* the screen grouping is "harder" as there is more variation in the types of screen pairs present. Therefore, our screen similarity model performs better for detecting similarity between two screens which is not surprising since the model architecture is designed around the same purpose, while the screen transition model can predict the transitions between consecutive screens.

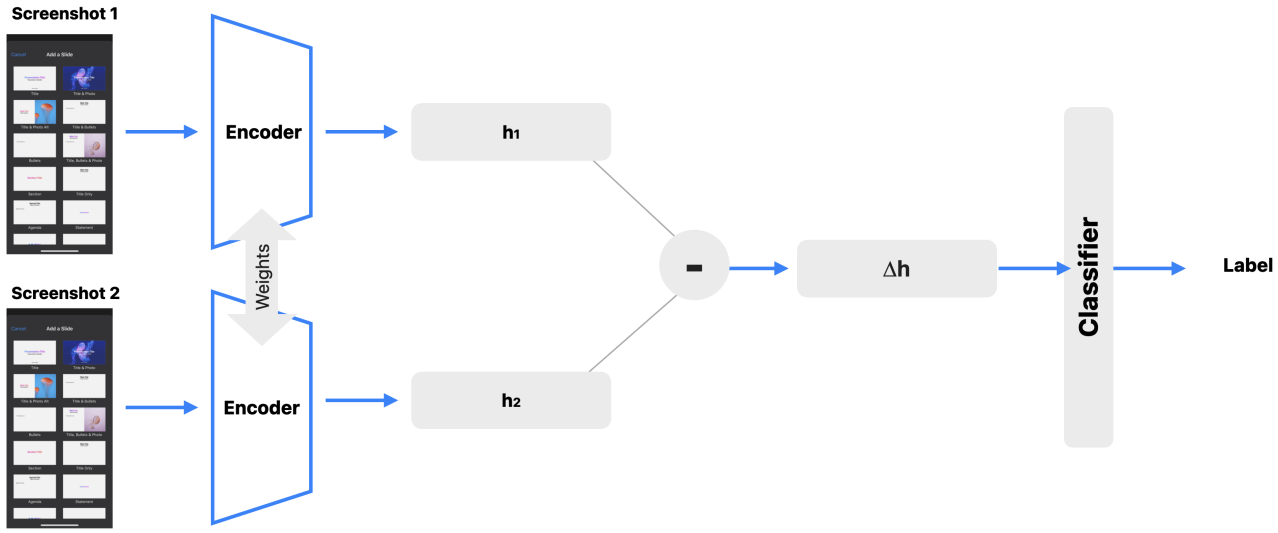

Figure 6: Screen transition model that classifies the transition type between two screens. Two screenshots are featurized using a CNN-based encoder. The resulting embedding vectors (h_1, h_2) are subtracted from each other to produce a difference vector Δh, which is used to predict transition type.

Table 2: Transition prediction performance for our Screen Transition model. For a reduced set of transitions (5), we achieve an average F1 score of 0.71, and we achieve an average F1 score of 0.65 for all transitions (11).

All Transition Types	Open	Close	Reduced Transition Types	Open	Close
App Dialogue	0.77	0.65	Dialogues	0.65	0.75
Bottom Dialogue	0.65	0.67			
System Dialogue	0.58	0.66			
Notification	0.64	0.64			
Keyboard	0.68	0.58	Keyboard	0.75	0.68
Scroll	0.58		Scroll	0.70	

6 DISCUSSION

Our results show that our models outperform existing models on the screen similarity task. This is not necessarily surprising, as most of the baseline algorithms (*e.g.*, image and perceptual hashing) are designed to detect image similarity more generally and are not trained on user interfaces. We should expect methods trained with user interface data to do better, as identifying that two screens are the same may require complex understanding of the semantics of an interface, such as understanding that different UI content retains the same layout of a previous screen or that a portion of the screen has changed position due to scrolling. Still, we see that our transformer-based method seems to learn these UI details better than the triplet-loss-trained embedding approach of TensorFlow Similarity. We hypothesize that our transformer-based model architecture (Similarity Transformer) is well-suited for our task, since it jointly encodes both screens (as opposed to independently computing an embedding for each), allowing it to identify corresponding elements through attention.

Our results also empirically confirm our intuitive notion that our two tasks are related, but the screen similarity task is harder than the screen transition task. This is expected because there are constraints on how a screen can change within a sequence. For example, a sequence of scrolling screenshots will likely retain some similar elements between consecutive screenshots, but in the grouping task the algorithm is expected to identify that an initial screen is the same as a screen scrolled several pages down, potentially showing very different content. We see this in our model training, where our models trained on the screen similarity task perform even better on the screen transition task, while performance lags and degrades in the opposite case of applying a trained transition model to the similarity task. This suggests different applications of screen similarity (*e.g.*, "fingerprinting" a UI v.s. processing an interaction trace) may necessitate adapting the data collection procedures, model architectures, and training strategies presented in this paper.

There are some limitations to our work. First is our dataset, which contains only 1,110 apps, many fewer than the millions reported to be available in the App Store. The apps in our dataset

also vary in terms of size and crawl length, meaning that the data is likely unbalanced in various ways. In some cases, an app may have few screens or few screens are recorded because the crawl worker was unable to get past the login screen. In other cases, motivated crawl workers may have found 20 or more unique screens in some apps. We also know that some transitions, particularly scrolling and notifications, are relatively uncommon in our dataset.

We also must question how well inexperienced annotators, who likely had no experience designing or building UIs, were at identifying similar screens. We spent significant effort to train annotators to identify screens as being similar, or not, according to our application's needs. We found that within-group agreement was high, suggesting that our annotators understood our instructions and were quite consistent. At the same time, we saw issues with our QA team, who wanted to group screens into a smaller number of groups than the annotators overall. As result, we used a smaller dataset for training and testing than we might have otherwise, because we chose to focus on data where all groups agreed. With improved annotation, and more agreement between the regular annotators and the QA team, we might expect our results to improve.

To further improve our models, we plan to collect and annotate additional data. We will also improve our annotation methods, particularly in QA for screen grouping, to ensure that we get better quality labels for a larger percentage of our data. With more data and improved annotation, we believe we can train models with greater performance.

We envision integrating our models into different applications such as automated crawling to investigate how they perform on real-world tasks. The automated crawler can use the screen similarity model to identify whether the current screen is the same as any previous screen that the crawler has encountered, so it can focus its efforts on unexplored portions of the app. Another application of understanding screen relationships is automated testing. For this type of application, the screen similarity model can be used to verify if the current screen that is viewed is the same as the recorded screen. In addition, screen transition model can help identify different types of events that might happen during automated testing. Finally, understanding screen relationships can be integrated into other applications such as automated UI testing, app understanding, automated playback of previously recorded interactions, and accessibility evaluation of apps.

7 CONCLUSION

We have examined two problems in this paper: the screen similarity problem requires identifying similar screens from a collection screenshots from an app, whereas the screen transition problem requires only identifying the type of transition events that might occur in an ordered sequence of screenshots from an user interaction trace. Our models achieve reasonable performance on both tasks, especially compared to a set of baseline algorithms. We expect in the future that these models will improve the performance of app crawlers and app automation systems.

ACKNOWLEDGMENTS

We thank our reviewers for their feedback which helped improve this paper.

REFERENCES

[1] (accessed)2021. Image Hash Library. Available at https://github.com/JohannesBuchner/imagehash.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59. https://doi.org/10.1109/MS.2014.55

[3] Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. 2021. VINS: Visual Search for Mobile User Interface Design. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–14.

[4] Elie Bursztein, James Long, Shun Lin, Owen Vallis, and Francois Chollet. 2021. TensorFlow Similarity: A Usable, High-Performance Metric Learning Library. *Fixme* (2021).

[5] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery dc: Design search and knowledge discovery through auto-created gui component gallery. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–22.

[6] Jia Chen, Ge Han, Shanqing Guo, and Wenrui Diao. 2018. FragDroid: Automated User Interface Interaction with Activity and Fragment Analysis in Android Applications. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 398–409. https://doi.org/10.1109/DSN.2018.00049

[7] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. StoryDroid: Automated Generation of Storyboard for Android Apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 596–607. https://doi.org/10.1109/ICSE.2019.00070

[8] Agnese Chiatti, Dolzodmaa Davaasuren, Nilam Ram, Prasenjit Mitra, Byron Reeves, and Thomas Robinson. 2019. Guess What's on my Screen? Clustering Smartphone Screenshots with Active Learning. *arXiv preprint arXiv:1901.02701* (2019).

[9] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology (UIST '17)*.

[10] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 767–776. https://doi.org/10.1145/2984511.2984581

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. 2014. Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications. *arXiv preprint arXiv:1402.4826* (2014).

[13] Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, Vol. 2. IEEE, 1735–1742.

[14] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. 204–217.

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

[16] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/3236024.3236055

[17] Ronghang Hu, Amanpreet Singh, Trevor Darrell, and Marcus Rohrbach. 2020. Iterative answer prediction with pointer-augmented multimodal transformers for textVQA. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9992–10002.

[18] Forrest Huang, John F. Canny, and Jeffrey Nichols. 2019. *Swire: Sketch-Based User Interface Retrieval.* Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3290605.3300334

[19] Zexun Jiang, Ruifeng Kuang, Jiaying Gong, Hao Yin, Yongqiang Lyu, and Xu Zhang. 2018. What Makes a Great Mobile App? A Quantitative Study Using a New Mobile Crawler. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 222–227. https://doi.org/10.1109/SOSE.2018.00037

[20] Luis A. Leiva, Asutosh Hota, and Antti Oulasvirta. 2020. Enrico: A Dataset for Topic Modeling of Mobile UI Designs. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services* (Oldenburg, Germany) *(MobileHCI '20)*. Association for Computing Machinery, New York, NY, USA, Article 9, 4 pages. https://doi.org/10.1145/3406324.3410710

[21] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. 2021. *Screen2Vec: Semantic Embedding of GUI Screens and GUI Components.* Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3411764.3445049

[22] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C).* IEEE, 23–26.

[23] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. 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).* 1070–1073. https://doi.org/10.1109/ASE.2019.00104

[24] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. DECAF: Detecting and characterizing ad fraud in mobile apps. In *11th USENIX symposium on networked systems design and implementation (NSDI 14).* 57–70.

[25] Chien-Hung Liu, Woei-Kae Chen, and Shu-Hang Ho. 2018. NFS: An Algorithm for Avoiding Restarts to Improve the Efficiency of Crawling Android Applications. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 02. 69–74. https://doi.org/10.1109/COMPSAC.2018.10205

[26] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *The 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18).* ACM, New York, NY, USA, 569–579. https://doi.org/10.1145/3242587.3242650

[27] Luka Malisa, Kari Kostiainen, and Srdjan Capkun. 2017. Detecting mobile application spoofing attacks by leveraging user visual similarity perception. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy.* 289–300.

[28] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015), 91–99.

[29] Amanpreet Singh, Vivek Natarajan, Meet Shah, Yu Jiang, Xinlei Chen, Dhruv Batra, Devi Parikh, and Marcus Rohrbach. 2019. Towards vqa models that can read. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 8317–8326.

[30] Hao Tan and Mohit Bansal. 2019. Lxmert: Learning cross-modality encoder representations from transformers. *arXiv preprint arXiv:1908.07490* (2019).

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

[32] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Information and Software Technology* 110 (2019), 139–155.

[33] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. Vet: Identifying and Avoiding UI Exploration Tarpits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021).* Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/3468264.3468554

[34] Jason Wu, Xiaoyi Zhang, Jeffrey Nichols, and Jeffrey P. Bigham. 2021. Screen Parsing: Towards Reverse Engineering of UI Models from Screenshots. In *Proceedings of the 2021 ACM Symposium on User Interface Software & Technology (UIST).* Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3472749.3474763

[35] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-duplicate detection in web app model inference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* 186–197.

[36] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering*, Vittorio Cortellessa and Dániel Varró (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 250–265.

[37] Shengcheng Yu, Chunrong Fang, Zhenfei Cao, Xu Wang, Tongyu Li, and Zhenyu Chen. 2021. Prioritize Crowdsourced Test Reports via Deep Screenshot Understanding. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 946–956.

[38] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, et al. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* 1–15.

[39] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology.* 609–621.

[40] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G.J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* 128–139. https://doi.org/10.1109/ICSE.2019.00030