Learning Design Semantics for Mobile Apps

Thomas F. Liu[1] Mark Craft[1] Jason Situ[1] Ersin Yumer[2] Radomir Mech[2] Ranjitha Kumar[1]
[1]University of Illinois at Urbana-Champaign [2]Adobe Systems Inc.
{tfliu2,mscraft2,junsitu2,ranjitha}@illinois.edu, meyumer@gmail.com, rmech@adobe.com

ABSTRACT
Recently, researchers have developed black-box approaches to mine design and interaction data from mobile apps. Although the data captured during this *interaction mining* is descriptive, it does not expose the *design semantics* of UIs: what elements on the screen *mean* and how they are used. This paper introduces an automatic approach for generating semantic annotations for mobile app UIs. Through an iterative open coding of 73k UI elements and 720 screens, we contribute a lexical database of 25 types of UI components, 197 text button concepts, and 135 icon classes shared across apps. We use this labeled data to learn code-based patterns to detect UI components and to train a convolutional neural network that distinguishes between icon classes with 94% accuracy. To demonstrate the efficacy of our approach at scale, we compute semantic annotations for the 72k unique UIs in the Rico dataset, assigning labels for 78% of the total visible, non-redundant elements.

Author Keywords
Design semantics; mobile app design; machine learning

INTRODUCTION
The ubiquity of mobile apps in everyday life — and their availability in centralized app repositories — make them an attractive source for mining digital design knowledge [1, 24]. Recently, Deka et al. introduced *interaction mining*, a black-box approach for capturing design and interaction data while an Android app is being used [9]. The data captured during interaction mining exposes a UI's screenshot; the elements it comprises along with their render-time properties (i.e., Android view hierarchy); and the interactions performed on the screen along with their connections to other UI states in the app. This data provides a near-complete specification of a UI that is often sufficient to reverse engineer it, but it fails to expose the *semantics* of UIs: what elements on the screen *mean* and how users interact with them to accomplish goals.

UIST '18, October 14–17, 2018, Berlin, Germany
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-5948-1/18/10. . . 15.00
DOI: https://doi.org/10.1145/3242587.3242650

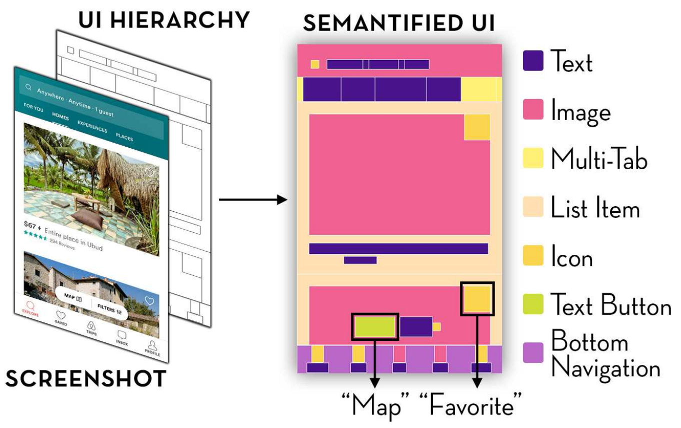


Figure 1: This paper introduces a code- and vision-based approach for adding semantic annotations to the elements comprising a mobile UI. Given a UI's screenshot and view hierarchy, we automatically identify 25 UI component categories, 197 text button concepts, and 99 icon classes.

This paper presents an automated approach for generating semantic annotations for mobile UI elements, given a screenshot and view hierarchy (Figure 1). These annotations identify both the *structural* roles (e.g., image content, bottom navigation) and the *functional* ones (e.g., login button, share icon) that elements play in the UI's design. To develop this approach, we first generated a lexical database of UI components and UX concepts (i.e., text buttons and icons) that are shared across apps through an iterative open coding of 73k UI elements and 720 screens. Then, we leveraged this database to learn code-based patterns to detect different components, and trained a convolutional neural network (CNN) to distinguish between icon classes.

To bootstrap this lexical database, we referenced popular design libraries and app-prototyping tools to create a vocabulary of UI components and UX concepts. We refined and augmented this vocabulary through unsupervised clustering and iterative coding of more than 73k elements sampled from the Rico dataset, which comprises interaction mining data from 9.7k Android apps [8]. Through this process, we identified 25 types of UI components, 197 text button concepts, and 135 icon classes. The resultant database also exposes icon and text-button synonym sets related to each UX concept, creating links between the visual and textual elements used in digital design.

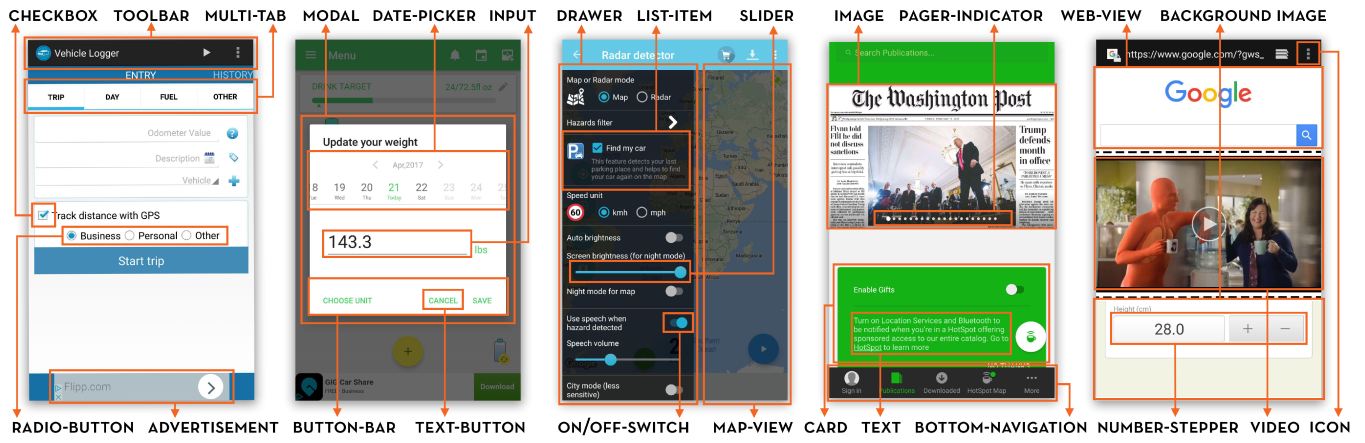


Figure 2: Screenshots illustrating the 25 UI component categories we identified through an open iterative coding of 720 UI screens from the Rico dataset.

To annotate mobile UIs, we employ a code- and vision-based approach that leverages the labeled data generated in the creation of the lexical database. We identified a set of textual and structural patterns in the view hierarchies that were predictive of each component class except for icons, which are difficult to distinguish from other images via static analysis. To identify icons, we first trained a convolutional neural network (CNN) which distinguishes between 99 icon classes with 94% accuracy. We pass the activation vectors produced by the CNN through a Gaussian mixture model trained to detect icon *anomalies*, or image inputs that do not belong to any of the 99 icon classes. The CNN with anomaly detection correctly predicts whether an image is an icon — and if so, the class to which it belongs — 90% of the time.

To demonstrate the utility of the approach at scale, we compute semantic annotations for the 72k unique UIs in the Rico dataset.[1] After removing hidden and redundant elements from Rico's view hierarchies, we achieve 78% element coverage with our technique. As a final contribution, we train an autoencoder over the annotated screenshots, illustrating how *element-level* annotations can be used to learn *screen-level* semantic embeddings.

CREATING A LEXICAL UI/UX DATABASE

Semantic models of design explain how low-level features such as colors, shapes, and interactivity communicate high-level concepts that affect how users perceive and use artifacts [5, 14, 16]. In this paper, we categorize UI components to provide semantics that describe the *structural* roles (e.g., image content, bottom navigation) elements play on a screen. Similarly, we analyze text buttons and icons to determine *functional* semantics: UX concepts that describe how elements and screens are used. Based on the UI/UX categories we identify and the labeled data generated in the process, we create a lexical database which enumerates a set of design semantics and maps them to implementations in existing apps.

[1]The UIs with semantically annotated elements are available at http://interactionmining.org/rico.

UI Components

To identify a set of UI component categories that frequently occur in Android apps, we referenced popular design tools and languages that expose component libraries such as Balsamiq [27] and Google's Material Design [4]. Using a consensus-driven, iterative approach, three researchers examined the component categories enumerated by these references, and merged together classes close in appearance and functionality. For example, "Date-Choosers" are merged together with DATE-PICKERS, and "Chips" with TEXT BUTTONS.

We also performed an iterative open coding of 720 screens from Rico, or approximately 1% of the dataset. We created this UI coding set by randomly sampling 30 apps from Rico's 24 app categories, and then randomly sampling one screen from each app. Three researchers from our team independently coded the screen's elements, noting any component types that were not part of the initial vocabulary. After the initial coding, the researchers met and discussed discrepancies and the set of new component categories until consensus was reached.

This process yielded 25 Android components: ADVERTISEMENT, BACKGROUND IMAGE, BOTTOM NAVIGATION, BUTTON BAR, CARD, CHECKBOX, DATE PICKER, DRAWER, ICON, IMAGE, INPUT, LIST ITEM, MAP VIEW, MODAL, MULTI-TAB, NUMBER STEPPER, ON/OFF SWITCH, PAGER INDICATOR, RADIO BUTTON, SLIDER, TEXT, TEXT BUTTON, TOOLBAR, VIDEO, and WEB VIEW. We provide a example of each component in Figure **??**.

UX Concepts

To construct a set of *functional* semantic concepts, we further analyzed TEXT BUTTON and ICON elements, since they indicate possible interactions on a screen (e.g., login button, search icon) [3]. We mine UX concepts directly from the text buttons by extracting and clustering the text contained within them. To mine UX concepts from icons, we first identify classes of visually similar icons through an iterative coding of 73k elements. Then we extract words (i.e., concepts) which describe each visual class by analyzing developer-defined code properties of its examples.

CONCEPT (#)	BUTTON TEXT STRINGS
no (5636):	no, no thanks, decline, disagree, reject, deny, refuse
login (5632):	login, sign in, log in, sign in with google, sign in with facebook, log in with facebook, log in with google, connect with facebook, login with facebook, log in with email, log in with twitter, sign in with email
back (5479):	back, cancel, return
go (5425):	go, start, begin, get started, proceed, start now
ok (5069):	ok, got it, okay, ok got it
all (4309):	all, view all, see all, clear all, select all
next (3308):	next
add (2672):	add, add to cart, add location, add to list, add to bag, add card
create (2585):	new, create, create account, new message, create an account, create new account
more (2544):	more, learn more, more info, load more, more apps, read more, more options
retry (2502):	retry, try again
skip (2428):	skip
set (2298):	set, set wallpaper, set up my news, set language, set as home
delete (2152):	delete, remove, clear, remove ads, clear all
facebook (1840):	facebook, sign in with facebook, sign up with facebook, login with facebook, log in with facebook, connect with facebook, continue with facebook
view (1766):	view, view all
book (1720):	book, book now
continue (1720):	continue, continue as guest, continue with facebook, continue shopping
list (1558):	list, add to list, mode list
save (1548):	save, save changes, save search
search (1407):	search, save search, search flights
finish (1342):	finish, done
agree (1249):	agree, yes, i agree, confirm
show (1233):	show, show password
share (1143):	share, share location
buy (1033):	buy, buy now, buy tickets
update (991):	update, upgrade, upgrade now, update now
edit (976):	edit, customize, refine, edit profile

Figure 3: The 28 most frequent UX concepts mined from buttons, identified through clustering 20,386 unique button text strings.

Text Buttons

Based on the text button examples we identified while categorizing UI components, we developed a heuristic for extracting text buttons from Android view hierarchies: any element whose `class` or `ancestors` property contains the string "button." Using this criteria, we extracted 130,761 buttons from the Rico dataset, and found 20,386 unique button text strings.

To mine UX concepts from these buttons, we filtered out button text comprising a single character or appearing in fewer than five apps, and clustered text strings that have substrings in common such as "retry" and "try again." We determined a UX concept (e.g., create, log in) for each cluster based on the most frequently occurring words in the set. Each unique text string can belong to more than one cluster. For example, "login with facebook" is part of both "login" and "facebook" concepts.

The research team verified the set of concept categories and button text mappings as a group. Team members discussed decisions to split or merge concept groups until consensus was reached. At the end of this process, we identified 197 text button concepts. Example concepts and related text button strings are shown in Figure 3.

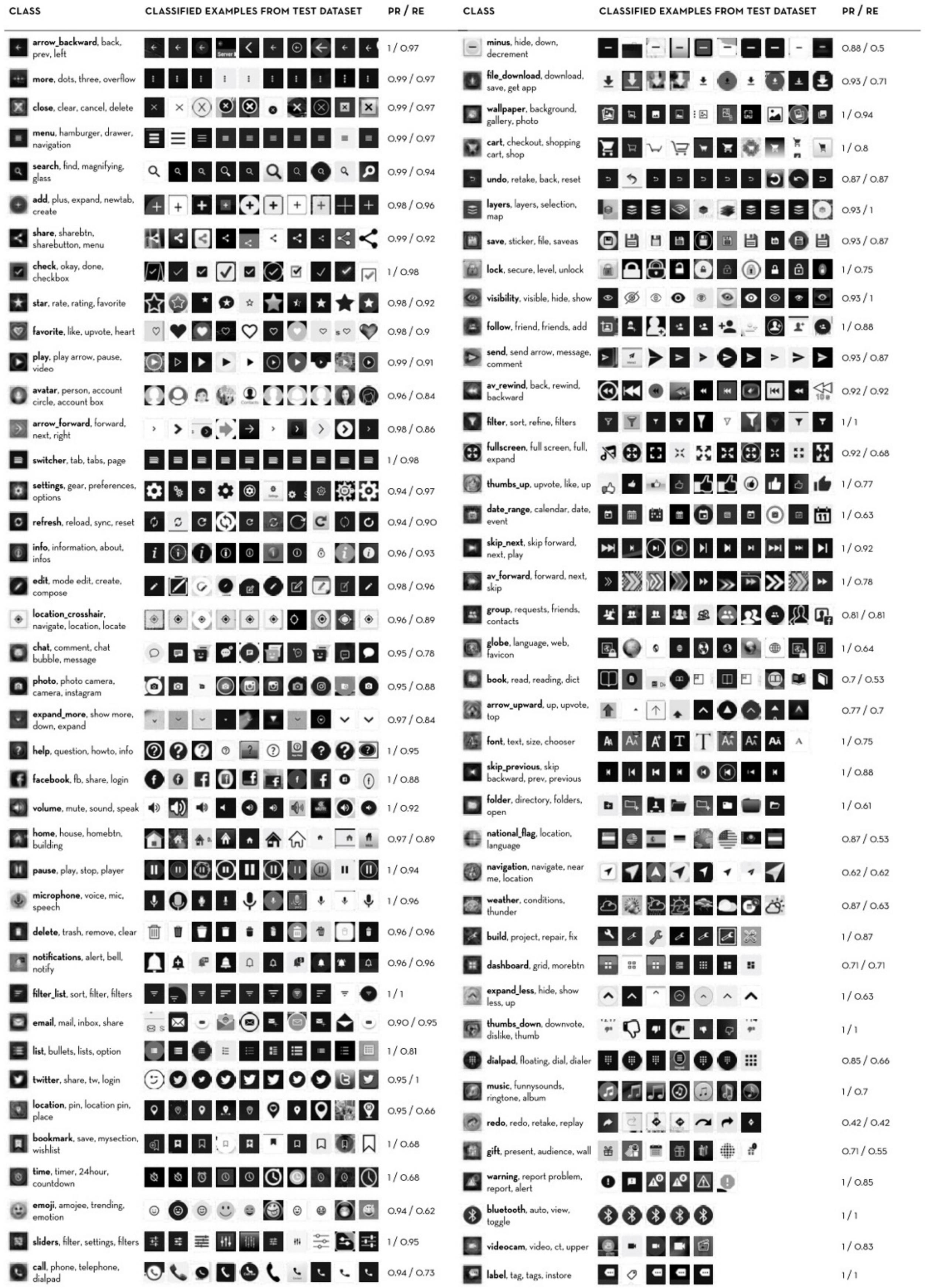
Icons

To create a taxonomy of icons, we extracted a large set of representative examples from the Rico dataset using the following heuristic: an image component that is `clickable` and `visible-to-user`, whose area is less than 5% of the total screen area, and whose aspect ratio is greater than 0.75 (smaller dimension divided by larger dimension). We use an element's `bounds` properties to compute its area and aspect ratio, and also to automatically crop the element from the screenshot.

Using this conservative heuristic — and after removing duplicate components from each app — we identified a set of 73,449 potential icons: small, squarish images. Through an iterative open coding of this set, we determined 135 icon classes that frequently occur in the Rico dataset.

We built two web applications to facilitate high-quality and efficient coding of icons. The first application allowed us to quickly assign classes to icons by presenting researchers with unlabeled images and a lexicon of categories bootstrapped from Google's Material icon set [12]. Researchers assigned labels using an input field that autocompleted on categories already in the lexicon (including 'not an icon'), and created new categories when necessary.

The second application enabled researchers to quickly review and refine categories and the elements assigned to them as a group. As part of this process, we identified distinguishing features for visually-similar icons to remove ambiguity. For example, we require that *skip_next* icons have a vertical line after the arrows to distinguish them from the visually-similar *av_forward* ⏩ class. We assigned 78% of the potential icons to 135 classes, determined that 11,205 images did not actually represent icons, and labeled the 13,108 remaining candidates as too niche to belong to a icon category.



CLASS	CLASSIFIED EXAMPLES FROM TEST DATASET	PR / RE	CLASS	CLASSIFIED EXAMPLES FROM TEST DATASET	PR / RE
arrow_backward, back, prev, left		1 / 0.97	**minus**, hide, down, decrement		0.88 / 0.5
more, dots, three, overflow		0.99 / 0.97	**file_download**, download, save, get app		0.93 / 0.71
close, clear, cancel, delete		0.99 / 0.97	**wallpaper**, background, gallery, photo		1 / 0.94
menu, hamburger, drawer, navigation		0.99 / 0.97	**cart**, checkout, shopping cart, shop		1 / 0.8
search, find, magnifying, glass		0.99 / 0.94	**undo**, retake, back, reset		0.87 / 0.87
add, plus, expand, newtab, create		0.98 / 0.96	**layers**, layers, selection, map		0.93 / 1
share, sharebtn, sharebutton, menu		0.99 / 0.92	**save**, sticker, file, saveas		0.93 / 0.87
check, okay, done, checkbox		1 / 0.98	**lock**, secure, level, unlock		1 / 0.75
star, rate, rating, favorite		0.98 / 0.92	**visibility**, visible, hide, show		0.93 / 1
favorite, like, upvote, heart		0.98 / 0.9	**follow**, friend, friends, add		1 / 0.88
play, play arrow, pause, video		0.99 / 0.91	**send**, send arrow, message, comment		0.93 / 0.87
avatar, person, account circle, account box		0.96 / 0.84	**av_rewind**, back, rewind, backward		0.92 / 0.92
arrow_forward, forward, next, right		0.98 / 0.86	**filter**, sort, refine, filters		1 / 1
switcher, tab, tabs, page		1 / 0.98	**fullscreen**, full screen, full, expand		0.92 / 0.68
settings, gear, preferences, options		0.94 / 0.97	**thumbs_up**, upvote, like, up		1 / 0.77
refresh, reload, sync, reset		0.94 / 0.90	**date_range**, calendar, date, event		1 / 0.63
info, information, about, infos		0.96 / 0.93	**skip_next**, skip forward, next, play		1 / 0.92
edit, mode edit, create, compose		0.98 / 0.96	**av_forward**, forward, next, skip		1 / 0.78
location_crosshair, navigate, location, locate		0.96 / 0.89	**group**, requests, friends, contacts		0.81 / 0.81
chat, comment, chat bubble, message		0.95 / 0.78	**globe**, language, web, favicon		1 / 0.64
photo, photo camera, camera, instagram		0.95 / 0.88	**book**, read, reading, dict		0.7 / 0.53
expand_more, show more, down, expand		0.97 / 0.84	**arrow_upward**, up, upvote, top		0.77 / 0.7
help, question, howto, info		1 / 0.95	**font**, text, size, chooser		1 / 0.75
facebook, fb, share, login		1 / 0.88	**skip_previous**, skip backward, prev, previous		1 / 0.88
volume, mute, sound, speak		1 / 0.92	**folder**, directory, folders, open		1 / 0.61
home, house, homebtn, building		0.97 / 0.89	**national_flag**, location, language		0.87 / 0.53
pause, play, stop, player		1 / 0.94	**navigation**, navigate, near me, location		0.62 / 0.62
microphone, voice, mic, speech		1 / 0.96	**weather**, conditions, thunder		0.87 / 0.63
delete, trash, remove, clear		0.96 / 0.96	**build**, project, repair, fix		1 / 0.87
notifications, alert, bell, notify		0.96 / 0.96	**dashboard**, grid, morebtn		0.71 / 0.71
filter_list, sort, filter, filters		1 / 1	**expand_less**, hide, show less, up		1 / 0.63
email, mail, inbox, share		0.90 / 0.95	**thumbs_down**, downvote, dislike, thumb		1 / 1
list, bullets, lists, option		1 / 0.81	**dialpad**, floating, dial, dialer		0.85 / 0.66
twitter, share, tw, login		0.95 / 1	**music**, funnysounds, ringtone, album		1 / 0.7
location, pin, location pin, place		0.95 / 0.66	**redo**, redo, retake, replay		0.42 / 0.42
bookmark, save, mysection, wishlist		1 / 0.68	**gift**, present, audience, wall		0.71 / 0.55
time, timer, 24hour, countdown		1 / 0.68	**warning**, report problem, report, alert		1 / 0.85
emoji, amojee, trending, emotion		0.94 / 0.62	**bluetooth**, auto, view, toggle		1 / 1
sliders, filter, settings, filters		1 / 0.95	**videocam**, video, ct, upper		1 / 0.83
call, phone, telephone, dialpad		0.94 / 0.73	**label**, tag, tags, instore		1 / 1

Figure 4: The 80 most frequent icon classes, identified through an iterative open coding of 73k elements from the Rico dataset. Given the labeled data for each icon class, we compute an "average icon" and a set of related text concepts. For each icon class, we also present precision/recall metrics from a CNN trained to distinguish between 99 common categories, and 10 predicted examples from the test set.

To map icons to UX concepts, we mine the set of words used by developers to characterize each icon class. While icons are concise, the same visual representation can have different meaning depending on the context (i.e., *polysemy*) or have a meaning that may not be immediately apparent [23]. To understand an icon category's range of meanings, we extracted the `resource-id` property associated with the examples for that class. We split each `resource-id` on underscores or via camel-case parsing, and performed tf-idf to identify the substrings that are characteristic of each class. We use these words sets — which are similar to the synonym sets exposed by WordNet [20] — to map icons to text concepts (Figure 4).

Our analysis reveals not only what developers call each icon, but also how the icons are used: for example, *national_flag* 🌐 is used for both location and language purposes. In fact, the lexical database can itself be used as a design resource to understand how button text and visually distinct icons relate to UX concepts. It reveals whether concepts are most often expressed as text or icons, and if there are visually distinct icons that have similar meaning (for example, *add* ⊞ and *edit* ◩ can both signify the concept of *creation*). We found that some concepts are predominantly text-based (*log in*, *sign in*), some are purely graphical (*navigation* ▲ , *gift* 🎁), and identified 94 that are expressed both as text buttons and icons.

AUTOMATING UI/UX DESIGN SEMANTICS

Having created a lexical database that enumerates a set of design semantics, we develop an automatic approach for annotating mobile UIs with them. Similar to the way ImageNet [10] used WordNet to organize images, we leverage the vocabulary provided by the lexical database and the set of labeled examples to learn how to automatically detect different components and concepts. Given an Android UI's screenshot and view hierarchy, we are able to automatically identify 25 UI component categories, 197 text button concepts, and 99 classes of icons.

Identifying Components

Prior work has shown that Android "widgets" are often constructed by grouping Android elements in regular ways [1, 24]. Shirazi et al. performed a static analysis of the XML files that define Android interfaces and exploited parent-child-element combinations to identify a common set of widgets [24]. For example, they identified two "TextView" elements within a "LinearLayout" as a common widget; however, they did not attribute any semantic meanings to these code patterns. Similarly, Alharbi et al. identified a small set of common UI components and describe how Android developers construct them [1].

Code-based heuristics

Using the labeled data provided by the lexical database, we discover and exploit similar code-based patterns in the view hierarchy to classify non-icon UI components. We determine the component type of an element primarily by examining its Android class and the classes it inherits from, exposed by its `class` and `ancestors` properties, respectively (Figure 5). For example, we identify IMAGES as elements whose class is "ImageView" or inherit from it. Some components have a few simple additional conditions: BACKGROUND IMAGE components, for instance, are IMAGES that cover most of the screen. To discover if characteristic class names exist for a UI component category, we extract the `class` and `ancestors` properties from labeled elements in its category, and run a tf-idf analysis over the set of strings.

In conjunction with class-based heuristics, structure-based patterns can be used to detect certain UI components. These rules examine an element's relation to its ancestors, descendants, or the view hierarchy itself.

UI COMPONENT	ASSOCIATED CLASS NAME
Advertisement:	AdView, HtmlBannerWebView, AdContainer
Background Image:	ImageView
Bottom Navigation:	BottomTabGroupView, BottomBar
Button Bar:	ButtonBar
Card:	CardView
Checkbox:	CheckBox, CheckedTextView
Drawer (Parent):	DrawerLayout
Date Picker:	DatePicker
Image:	ImageView
Image Button:	ImageButton, GlyphView, AppCompactButton, AppCompactImageButton, ActionMenuItemView, ActionMenuItemPresenter
Input:	EditText, SearchBoxView, AppCompatAutoCompleteTextView, TextView
List Item (Parent):	ListView, RecyclerBiew, ListPopUpWindow, TabItem, GridView
Map View:	MapView
Multi-Tab:	SlidingTab
Number Stepper:	NumberPicker
On/Off Switch:	Switch
Pager Indicator:	ViewPagerIndicatorDots, PageIndicator, CircleIndicator, PagerIndicator
Radio Button:	RadioButton, CheckedTextView
Slider:	SeekBar
Text Button:	Button, TextView
Toolbar:	ToolBar, TitleBar, ActionBar
Video:	VideoView
Web View:	WebView

Figure 5: We identify most components by their class or the classes they inherit from, and Drawers and List Items based on the class names of their parent nodes.

UI components such as LIST ITEM and DRAWER occur as children or descendants of specific types of nodes. To identify LIST ITEM, which can be assigned any Android class, we identify container classes that represent lists such as "ListView" and "RecyclerView," and then label the elements inside those containers as LIST ITEM. DRAWER is recognized in a similar way: the second child of a "DrawerLayout" element (the first child holds the elements below the drawer). Conversely, PAGER INDICATOR components can be identified through their children since they are often constructed via a generic layout container with multiple, small children representing the dots inside. MODAL components are elements that occur at the top of the view hierarchy, but have an area that is smaller than the size of the phone screen.

Finally, elements that have not been labeled anything else, but have a non-empty `text` view hierarchy property are labeled as TEXT components.

Results

Based on the outlined rules, we computed semantic labels for the elements in the Rico dataset. To compute the total number of elements in Rico which *could* have labels, we removed invisible, and redundant nodes that do not contribute to the UI screen [24]. In total, our approach can label 1,384,653 out of the 1,776,412 (77.95%) visible, non-redundant elements present in the Rico dataset, averaging 25 component annotations per UI.

Figure 6 shows the distribution of app categories for each of 16 UI components. From this distribution, we can make a number of observations. Health and fitness apps tend to use the NUMBER STEPPER component much more than other types of apps, perhaps because the entry of numerical information is a common activity within those apps. As one would expect, MAP VIEW predominantly occurs in weather, travel, and navigation apps. Interestingly, the BOTTOM NAVIGATION component is commonly used in social apps. Overall, TEXT, IMAGE, TEXT BUTTON, TOOLBARS, and LIST ITEM account for the majority of components.

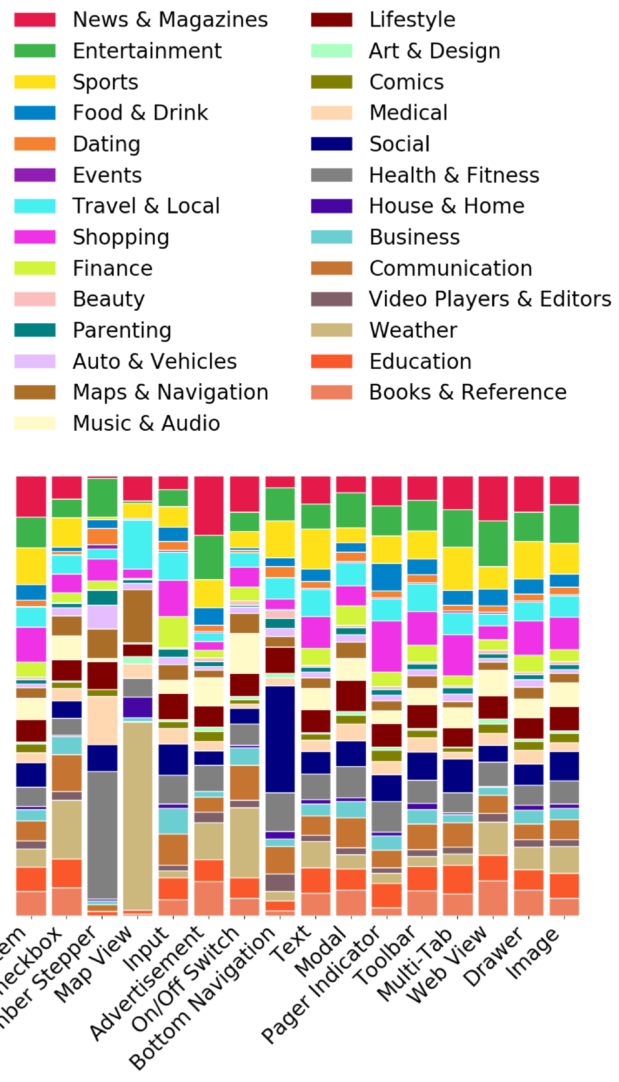


Figure 6: The distribution of Rico's 27 app categories for each of 16 UI components we identified.

Classifying Icons

Based on the labeled data, we could not find a code-based heuristic that distinguishes icons from images with high accuracy: the class names used to identify icons are not consistent. Therefore, we train a vision-based, deep learning pipeline that distinguishes between images and 99 icon classes, adapting a convolutional neural network (CNN) architecture [7] to multiclass icon classification.

Preprocessing

We leverage the icons we extracted and labeled during the creation of the lexical database for training. We merged classes with similar meaning or appearance to decrease ambiguity and increase support. For instance, the *more_horizontal* ⊟ and *more_vertical* ⊡ classes were combined into a single *more* class, since one is just a rotation of the other; *warning* ▲ , *error* ● , and *announcement* ❗ were similarly combined. We used only icon classes with more than 40 examples for training. After merging and filtering, we identified 99 classes in a training set of 49,136 icons.

We preprocessed the icon images in the training dataset using the following standard vision techniques:

Conversion to greyscale: We converted icons to greyscale to eliminate the effect of color on the convolutional neural network, since color is usually not a factor in the design and comprehensibility of an icon [15, 6]. Using greyscale images also reduces training time since it reduces the dimensionality of the problem.

Featurewise centering and standard deviation: The mean over the entire dataset was set to 0 and the standard deviation to 1. Samplewise centering and samplewise standard deviation normalization worsened results and were not used.

Translation: Images were randomly moved along the x- and y-axis by up to 10% of the image width.

ZCA whitening: Since icon meaning relies on shape-based features, we use ZCA whitening to boost edges [6].

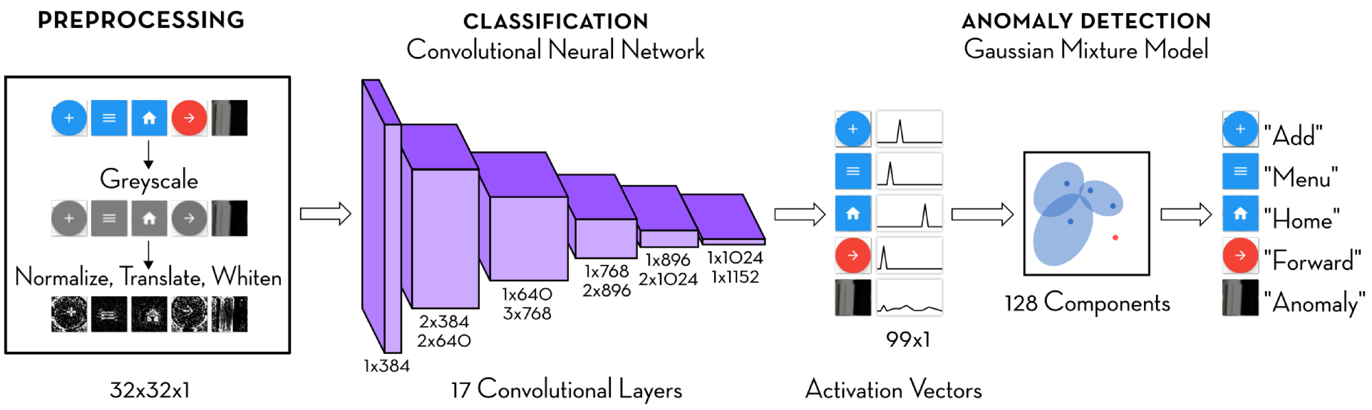


Figure 7: We pass all small, squarish images found in a UI's view hierarchy through a classification pipeline that distinguishes between 99 classes of icons and general image content using a CNN and anomaly detector, respectively. CNN stack labels denote layers × filters.

Model Architecture

The classification of small images is a well-studied problem in several domains. For example, MNIST is a set of handwritten digits commonly used for the training and evaluation of computer vision models [19]. The CIFAR-100 dataset consists of 32 × 32 images of animals, plants, vehicles, devices, and other real life objects [18].

We adapt the CNN architecture that currently performs the best on the CIFAR-100 dataset to multiclass icon classification since we have roughly the same number of classes [7]. Moreover, given that icons tend to be simpler and more abstract than natural images, we hypothesized that the representational power of the architecture was sufficient to learn the differences between the icon classes.

Our model comprises 17 convolutional layers, arranged into 6 stacks. After each stack, we applied max pooling with a stride and filter size of 2, and increased the dropout by 0.1 (the initial dropout rate is 0). The last two layers are fully connected, one with 512 neurons, and a final softmax activation layer with 99 neurons. We trained the network to minimize categorical cross entropy loss with RMSprop. All layers except the last used ELU activation.

Anomaly Detection

To distinguish between icons and images, the icon classification pipeline should be able to detect images that differ significantly from those in the icon dataset. We train an anomaly detector that flags anomalous activations of the CNN's last softmax layer.

We computed the activations for each image in the potential icon dataset that we had labeled, which included positive and negative examples of icons, and trained a Gaussian Mixture Model on these activations. Specifically, we used scikit-learn's GaussianMixture with 128 components and full covariance [25]. Figure 7 shows the icon classification pipeline in its entirety.

Results

We trained the CNN on 90% of the training examples for each icon class, and used 10% as a holdout test set. We achieve an overall accuracy of 99% on the training set, and 94% on the test set (Figure 8). The macro precision and recall metrics are the averages of the precision and recall of each class, respectively. Unlike accuracy, each class is weighed equally in these metrics.

Weak supervision from textual metadata has been useful for increasing the accuracy of semantic predictions in domains such as 3D modeling [28]. Therefore, we experimented with using textual metadata from the Android view hierarchy, such as associated text concepts mined from resource-ids: however, the additional training inputs did not increase performance. Keeping a vision-only approach also allows icon classification on platforms and digital domains without Android metadata.

Viewing anomaly detection as a binary classification problem, we are able to achieve 90% precision and 98% recall on valid icons, meaning that the anomaly detector discards only 2% of valid icons. On the set of anomalies, we achieve 94% precision and 78% recall.

The accuracy, precision, and recall metrics of the CNN combined with the anomaly detection GMM are displayed in Figure 8. The overall accuracy of the classifier on the test dataset decreases from 94% to 91%, and recall decreases from 86% to 79% because some valid icons are classified as anomalies. However, the precision of the CNN *increases* slightly from 87% to 91% since the anomaly detector often discards icons that the CNN has difficulty placing into one of the 99 classes.

	NO ANOMALY DETECTION		WITH ANOMALY DETECTION	
	TRAIN	TEST	TRAIN	TEST
ACCURACY	0.9949	0.9443	0.9712	0.9098
MACRO PRECISION	0.9946	0.8743	0.9783	0.9062
MACRO RECALL	0.9946	0.8573	0.9697	0.7910

Figure 8: Icon classification performance metrics for 99 common classes, with and without anomaly detection.

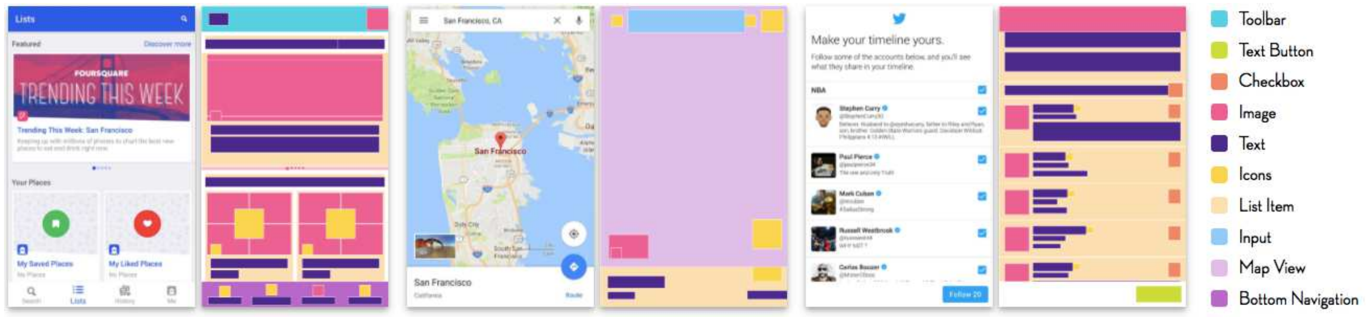


Figure 9: Our automated semantic approach applied to different mobile UIs. We extracted code-based patterns to semantically classify 24 out of the 25 types of UI components on a screen. To identify icons, we trained a convolutional neural network and anomaly detection pipeline which distinguishes between images and 99 common classes of icons.

Annotating Rico

We use the code-based patterns for detecting UI components and the icon classification pipeline to add semantic annotations to the 72k unique UIs in the Rico dataset. The automated approach iterates over all elements in a UI's view hierarchy, and uses code-based properties to identify different types of UI components and text button concepts. To identify icons, we pass all small, squarish images found in a UI's view hierarchy through the classification and anomaly detection pipeline.

Once we semantically label the elements in a view hierarchy, we also generate a semantic version of its screenshot and view hierarchy, which are useful representations for future data-driven applications (Figure 9). The annotated UIs for the Rico dataset are available for download at `http://interactionmining.org/rico`.

Training a Semantic Embedding

Deka et al. demonstrated how training an embedding using UI screens that encode image and text content could power example-based UI searches to recover visually similar UI screens [8]. We implement a similar application, using the semantic versions of the UI screenshots we computed over the Rico dataset to train a convolutional autoencoder.

An autoencoder is a neural network consisting of two components: an *encoder* that reduces the image down to a lower dimensional latent space, and a *decoder* that attempts to reconstruct the image from the encoding. By training an autoencoder to accurately reproduce images, the architecture learns a low-dimensional encoding representation for each one.

We downsampled the semantic screenshot images to 256×128. Our encoder consists of 4 convolutional layers arranged as 8×3, 16×3, 16×3, and 32×3 (filters $\times$ receptive fields). A max pooling layer of size and stride 2 is applied after every convolutional layer, resulting in a $32 \times 16 \times 32$ encoded representation. Our decoder consists of the encoding layers in reverse order, with upsampling layers instead of max pooling layers, and a final 3×3 layer to convert back to the original RGB input.

After training the autoencoder, we used the encoder to embed the Rico UIs into the learned latent space, and then inserted the embedded vectors into a ball tree [26] to run fast nearest neighbor searches (Figure 10).

We compare the nearest neighbor search results with the ones demonstrated by Deka et al. [8], which used screenshots encoding only text and image regions. Figure 10 shows search results for both embeddings side-by-side over the same set of queries. Our results demonstrate that autoencoders trained with finer *element-level* semantics learn better *screen-level* semantic embeddings.

LIMITATIONS AND FUTURE WORK

One limitation of this work is that many of the UI components we have identified are specific to Android apps. However, our vision-based classification pipeline is cross-platform, and researchers can use this approach to train models that detect other types of UI components found in other digital domains.

Another limitation is that the current CNN architecture does not perform well on icons with low support in the training set. As a result, we could not train a multiclass classifier that recognized all icon types that we originally identified. Future work could adapt one-shot image recognition techniques to train better models for detecting new icons and those with low support. These algorithms often exploit prior knowledge learned from classes with large support to predict classes with fewer training examples [17].

We hope that researchers leverage our semantic data to develop new data-driven design applications. Semantic annotations enable many kinds of design-based search interactions. Semantic embeddings similar to the one demonstrated in this paper can be trained to support example-based search over app screens: designers can query with screens they have designed to see which apps contain similar interactions.

Similarly, icon and text button classification can be used to enable efficient flow search over large datasets of interaction mining data. A user flow is a logical sequence of UI screens for accomplishing a specific task in an app. Prior work has shown that user flows can be identified in interaction traces by examining the UI elements that a user has interacted with [9]. For example, if a user taps on a "search" icon, that usually signifies the beginning of a search flow. Rico's 72k dataset of unique UIs comprises 14,495 click interactions, of which 9,461 involve text buttons or images. We are able to classify 57.5% of those interactions based on our set of 197 recognized buttons and 99 icon types, which can serve as a starting point for indexing flows.

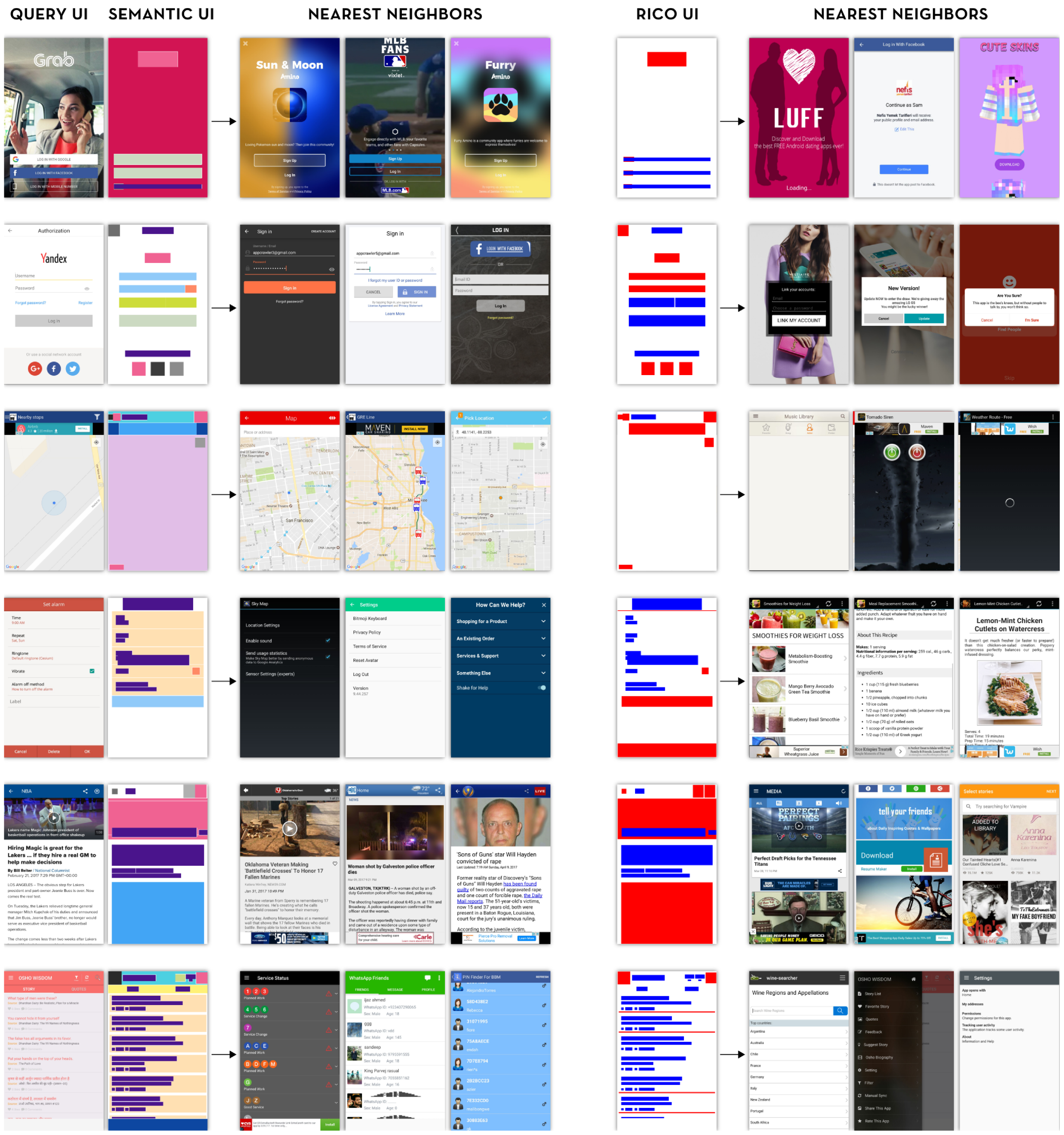


Figure 10: We train a autoencoder over the computed semantic screenshots, and compare nearest neighbor UI searches in this learned space to the results presented in Deka et al. [8]. These results demonstrate that autoencoders trained with finer *element-level* semantics learn better *screen-level* semantic embeddings.

Leveraging semantics to train generative models of design is possibly the most exciting avenue for future work. Researchers have proposed methods for automatically generating mobile UI code from screenshots [2, 21, 13, 22]. These approaches seem to only work for simple UIs, and do not exploit the view hierarchy present in Android apps.

By taking semantic components as input, more powerful probabilistic generative models of mobile UIs could be learned. Techniques such as generative adversarial networks and variational autoencoders could be used to build design tools that can *autocomplete* partial specifications of designs [11]. For example, the tool could recognize that a designer is creating a "login" screen based on the central placement of a login button, and suggest adding other elements such as "username" and "password" input fields. These tools could translate a set of semantic constraints specified by a designer (e.g., login button) into a cohesive visual layout and information architecture, as well as suggest UI elements that are missing from the design.

ACKNOWLEDGMENTS
We thank the reviewers for their helpful comments and suggestions. This work was supported in part by a research donation from Adobe, a Google Faculty Research Award, and NSF Grant IIS-1750563.

REFERENCES
1. Khalid Alharbi and Tom Yeh. 2015. Collect, decompile, extract, stats, and diff: Mining design pattern changes in Android apps. In *Proc. MobileHCI*.
2. Tony Beltramelli. 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. *arXiv preprint arXiv:1705.07962* (2017).
3. Elizabeth Boling, Joanne E Beriswill, Richard Xaver, Christopher Hebb, and others. 1998. Text labels for hypertext navigation buttons. *International Journal of Instructional Media* 25 (1998), 407.
4. Call-Em-All. 2018. Material-UI. (2018). `https://material-ui-next.com/`
5. Siddhartha Chaudhuri, Evangelos Kalogerakis, Stephen Giguere, and Thomas Funkhouser. 2013. Attribit: content creation with semantic attributes. In *Proc. UIST*.
6. Chun-Ching Chen. 2015. User recognition and preference of app icon stylization design on the smartphone. In *Proc. HCII*. Springer.
7. Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2015. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *CoRR* abs/1511.07289 (2015).
8. 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 *Proc. UIST*.
9. Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proc. UIST*.
10. Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proc. CVPR*.
11. Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Proc. NIPS*.
12. Google. 2017. Material Icons. (2017). `https://material.io/icons/`
13. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015).
14. Ali Jahanian, Shaiyan Keshvari, SVN Vishwanathan, and Jan P Allebach. 2017. Colors–Messengers of Concepts: Visual Design Mining for Learning Color Semantics. In *Proc. TOCHI*.
15. Charles J Kacmar and Jane M Carey. 1991. Assessing the usability of icons in user interfaces. *Behaviour & Information Technology* 10 (1991).
16. Shigenobu Kobayashi. 1991. *Color Image Scale*. Kosdansha International.
17. Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. 2015. Siamese neural networks for one-shot image recognition. In *Proc. ICML Deep Learning Workshop*.
18. Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.
19. Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).
20. George A Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38 (1995).
21. Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *arXiv preprint arXiv:1802.02312* (2018).
22. Siva Natarajan and Christoph Csallner. 2018. P2A: A Tool for Converting Pixels to Animated Mobile Application User Interfaces. In *Proc. MOBILESoft*.
23. Yvonne Rogers. 1989. Icons at the interface: their usefulness. *Interacting with Computers* 1 (1989).
24. Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In *Proc. EICS*.
25. scikit-learn developers. 2017a. Gaussian mixture models. (2017). `http://scikit-learn.org/stable/modules/mixture.html`

26. scikit-learn developers. 2017b. sklearn.neighbors.BallTree. (2017). http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html

27. Balsamiq Studios. 2018. basalmiq. (2018). https://balsamiq.com/

28. Li Yi, Leonidas J. Guibas, Aaron Hertzmann, Vladimir G. Kim, Hao Su, and Ersin Yumer. 2017. Learning Hierarchical Shape Segmentation and Labeling from Online Repositories. In *Proc. SIGGRAPH*.