Approximately 50% of development resources are devoted to user interface (UI) development tasks [9]. Occupying a large proportion of development resources, developing icons can be a time-consuming task, because developers need to consider not only effective implementation methods but also easy-to-understand descriptions. In this article, we present Auto-Icon+, an approach for automatically generating readable and efficient code for icons from design artifacts. According to our interviews to understand the gap between designers (icons are assembled from multiple components) and developers (icons as single images), we apply a heuristic clustering algorithm to compose the components into an icon image. We then propose an approach based on a deep learning model and computer vision methods to convert the composed icon image to fonts with descriptive labels, thereby reducing the laborious manual effort for developers and facilitating UI development. We quantitatively evaluate the quality of our method in the real-world UI development environment and demonstrate that our method offers developers accurate, efficient, readable, and usable code for icon designs, in terms of saving 65.2% implementing time.

CCS Concepts: • Human-centered computing → Empirical studies in accessibility;
Additional Key Words and Phrases: Code accessibility, icon implementation, neural networks

1 INTRODUCTION

A user interface (UI) consists of series of elements, such as text, colors, images, and widgets. Designers are constantly focusing on icons, as they are highly functional in a UI [11, 50, 52, 67]. One of the biggest benefits of icons is that they can be universal. For instance, by adding a red “X” icon to your UI design, users are informed that clicking this icon leads to the closure of a component. Furthermore, icons can make UIs look more engaging. For example, instead of using basic bullets
or drop-downs filled with words, a themed group of icons can capture instant attention from users. Consequently, icons become an elegant yet efficient way to communicate with and help guide the user through experience.

Despite of all these benefits, icons have three fundamental limitations in the day-to-day development environment, in terms of transition gap, rendering speed, and code accessibility. First, it is a challenging task for developers to implement icons from the design artifacts, as many designers follow their own design styles that highly differ from each other. For example, some of them design an icon with many small components but just one combined image in the icon implementation for optimizing the network traffic and caching. Such a gap adds the complexity and effort for developers to preprocess the design draft before icon implementation. Second, to ensure a smooth user interaction, UI should be rendered in under 16 ms [29, 30, 42], whereas an icon implemented as an image faces the slow rendering problem due to image download speed, image loading efficiency, and so on. These issues will directly affect the quality of the product and user experience, requiring more effort from developers to develop an advanced method to overcome the problem. Third, in the process of UI implementation, many developers directly import the icon resources from the design artifacts without considering the meaning of the content, resulting in a poor description/comment during coding. Different codes render the same visual effect to users, although it is different for developers to develop and maintain. A non-descriptive code increases the complexity and effort required for developers, as they need to look at the associated location of the UI to understand the meaning of the code.

This challenge motivates us to develop a proactive tool to address the existing UI development limitations and improve the efficiency and accessibility of code. Our tool, Auto-Icon+, involves four main features. First, to bridge the conceptual gap between icon design and icon implementation, we propose a heuristic machine learning technique to automatically aggregate the scattered icon components into clusters. The component clusters can then be composed into icon images, which can be applied in various downstream tasks related to improving the rendering efficiency (i.e., icon font, CSS clipping) and code accessibility (i.e., description, color). Second, to meet the requirement of efficient rendering, we develop an automated technique to convert the icon image to an icon font, which is a typeface font. Once the font is loaded, the icon will be rendered immediately without downloading the image resources, thereby reducing HTTP requests and improving the rendering speed. The icon font can further optimize the performance of rendering by adopting HTML5 offline storage. In addition, the icon font has other potential attributes that can facilitate UI development, such as being easy to use (i.e., use the CSS’s @fontface attribute to load the font) and flexible (i.e., capable of changing color, lossless scale). Third, understanding the meaning of icons is a challenging problem. There are numerous types of icons in the UIs. Icons representing the same meaning can have different styles and can be presented in different scales as shown in Table 1. In addition, icons are often not co-located with texts explaining their meaning, making it difficult to understand from the context. To offer easy access for developers to develop thorough understanding of the meaning of icons, we collect 100k icons from the existing icon sharing website Alibaba Iconfont [3], each associated with a label described by the designer. By analyzing the icons and labels, we construct 100 categories, such as “left,” “pay,” “calendar,” and “house.” We then train a deep learning classification model to predict the category of the icon as its description. The experiments demonstrate that our model, with an average accuracy of 0.87 in an efficient classification speed at 17.48 ms, outperforms the other deep learning based models and computer vision based methods. Third, to provide more accessibility to developers on the description of icon images, we also detect the primary color of icons by adopting HSV color space [95]. We refer to our mechanism tool Auto-Icon+ to build an intelligent support for developers in the real context of UI development, assisting in the development of standardized and efficient code.
To demonstrate the usefulness of Auto-Icon+, we carry out a user study to show if our tool for automatically converting an icon to a font with label descriptions can help provide more knowledge on code accessibility and accelerate UI development for developers. After analyzing the feedback of 10 professional developers with all positive responses on our mechanism tool, we found that the code for the icon generated by our tool can achieve better readability compared with the code manually written by professional developers. In addition, Auto-Icon+ has been implemented and deployed in the Alibaba Imgcook platform. The results demonstrate that our tool provides 84% usable code for icon designs in realistic development situations. Our contributions can be summarized as follows:

- We identify the fundamental limitations of existing UI development of icon images. The informal interviews with professional developers also confirm these issues qualitatively.
- To bridge the gap between icon design and icon implementation, we develop a machine learning based technique to compose the components of an icon in the design artifact.
- Based on the emerging 100 icon categories, we develop deep learning and computer vision based techniques for specifically converting the icon to a font with a label describing its meaning and color to provide developers code understanding.
- We conduct large-scale experiments to evaluate the performance of our tool Auto-Icon+ and show that our tool achieves good accuracy compared with baselines. The evaluation conducted with developers and tested on the real-world development platform demonstrates the usefulness of our tool.
- We contribute to the community by offering intelligent support for developers to efficiently implement icon designs that comply with code standardization.

2 RELATED WORKS

2.1 Transition Gap

UI design and implementation require a different mind-set and expertise. The former is performed by user experience designers and architects via design tools (e.g., Sketch [93], Photoshop [83]), whereas the latter is performed by developers via development tools (e.g., Android Studio [94], Xcode [7]). Existing research well supports these two phases, respectively [6, 12, 20, 36, 56, 96, 106, 107]. For example, Li et al. [58] provide a holistic representation of UI to help designers quickly build up a realistic understanding of the design space for an app feature and get design inspirations from existing UI designs. But few of them support effective transition from UI design artifacts to UI implementation.

Supporting this transition is challenging due to the gap between designers and developers and the complexity of design artifacts (see Figure 1). Some research lowers the transition gap by translating UI screenshots into UI implementation [2, 17, 21, 71, 76, 101]. Nguyen and Csallner [81] follow the mobile-specific heuristics and adopt a hybrid method based on Optical Character Recognition (OCR) and image processing to generate a static code from UI screenshots. Pix2Code [10] adopts an iterative encoder/decoder deep learning model consisting of a convolutional neural network (CNN) for extracting UI screenshots features and a LSTM decoder for generating the UI code tokens. However, these works are unclear if the UI implementations are realistic or useful from a developer’s point of view, as the implementation of the approaches are only validated on a small set of synthetic UI screenshots. It is difficult to judge how well the approaches would perform on real UI screenshots. In contrast, Auto-Icon+ is tailored for the design artifact driven development practice and is aimed at solving real-world UI development challenges, such as developers requiring manual effort to compose the icon from scattered components in the design artifact. There has also been both commercial and academic work related to design artifact development.
driven development for creating high-fidelity design artifacts [37, 73, 84], deploying real-device prototype environments [69, 70], and verifying design violations [75]. However, such tools and approaches tend to either impose too many restrictions on designers or do not allow for direct creation of code, thus the icon composition problem persists in practice. To address this, we adopt a machine learning approach to automatically agglomerate the scattered icon components into clusters through heuristic correlation coefficient to bridge the gap between icon design and icon implementation.

2.2 UI Rendering

Ensuring fast rendering speed is an essential part in UI development, since slow rendering creates a poor user experience. Many studies focus on improving rendering speed via reducing bugs [14, 47, 59, 80, 86]. In contrast, we focus on analyzing image displaying performance in UI rendering. There are a few related works in this domain. For example, Systrace [5] is a tool that allows developers to collect precise timing information about UI rendering on devices. However, it does not provide any suggestions for improvement. To address this problem, many studies introduce reliable approaches to improve rendering efficiency, such as image resizing based on pattern-based analysis [63] and a manual image resource management based on resource leakage analysis [102]. Gao et al. [41] implement a system called DRAW, which aims to reveal UI performance problems in an application such as excessive overdraw and slow image components detection. With the suggestion of the image displaying performance analysis by DRAW, developers can manually improve the rendering performance of slow image display. Although these works offer image management suggestions to developers to achieve better rendering performance, they still need to be improved manually. In contrast, we propose an image conversion technology based on computer vision and graphic algorithms to convert icons into font types for achieving faster UI rendering.

2.3 Code Accessibility

Digital devices such as computers, mobile phones, and tablets are widely used. To ensure the quality of software, many research works have been conducted [13, 40, 65]. Most of these works focus on the functionality and usability of apps such as GUI design [16, 19, 21, 35, 64, 104, 112], GUI animation linting [109, 110], localization [34, 98, 104], privacy and security [22, 23, 26, 33], and performance [62, 111]. A few research works are related to accessibility issues. Some works in the human-computer interaction area have explored the accessibility issues of mobile apps [18, 51, 74, 97, 105]. In these works, the lack of description in image-based components in UI is commonly regarded as an important accessibility issue. For instance, Harrison et al. [44] establish an initial “kineticicon vocabulary” containing a set of 39 kinetic behaviors for icon images, such as spin, bounce, and running. Ross et al. [87] identify some common labeling issues in Android apps via analyzing the icon image labeling. With a crowdsource method, Zhang et al. [108] annotate GUI elements without content description. However, these works still require support from developers. Due to the increasingly developed CNN technologies, dramatic advances appear in the field of image classification that are applied to automatically annotate tags for images. Chen et al. [15] analyze the tags associated with the whole GUI artwork collected from Dribbble and emerge with a vocabulary that summarizes the relationship between the tags. Based on the vocabulary, they adopt a classification model to recommend the general tags in the GUI, such as “sport” and “food.” Different from their work, we predict more fine-grained categories, such as “football” and “burger.” In addition, they focus on predicting the categories of the whole UI, which is subjective to human perception, but the categories of small icons are usually more intuitive. A similar work to ours is the icon sensitive classification by Xiao et al. [103]. They utilize traditional computer vision techniques like SIFT and FAST to extract the features of icons and classify icons into eight categories through calculating
their similarity. After the systematically investigation of icons, we discover the fundamental limitations in icons discussed in Section 4.1 in terms of high cross-class similarity and small, transparent, and low contrast. These findings conflict with methods applied in their work, such as applying rotation to augment the dataset. Moreover, we show that the deep learning model is fruitful for the icon classification problem over the tradition computer vision technique in Section 6.2.3. In our work, according to the characteristic of icons, we propose a deep learning model to automatically classify icons in a more fine-grained (100) category and also adopt a computer vision technique to detect its primary color.

3 PRELIMINARY STUDY

To better understand the challenges in the real-world UI development environment, we conducted an interview with 12 front-end developers from big companies. Two authors first developed the interview protocol and then conducted pilot studies with two participants. Based on the pilot studies, we refined the interview protocol and conducted 10 formal interviews. The average length of these interviews was 20 minutes. We started with general questions, such as questions about working years, workload of development, and number of projects developed. Then, we asked the interviewees how they developed the code for icon images. We particularly asked what motivated them to adopt the approach, whether they revised the implementation, what approaches could achieve the same effect, what they perceived as the impact of the implementation, how the implementation behaved in the process of development, and if there was any difference in UI development between personal projects and company tasks.

3.1 Research Question 1: Do Developers Implement Icons Directly from the Design Artifacts?

Once the design artifacts are completed by designers, they are handed off to development teams who are responsible for implementing the designs in code. Figure 1 shows an example of the design artifact handed off to developers. The design artifacts are stored as archives containing JSON encoded layout and several binary assets, such as bitmap images, curves, and shapes [92]. The layout is structured as a cumulative hierarchy comprising a tree structure, starting with a single root node, where the spatial layout of the parent always encompasses contained child components. A discrete component with a corresponding set of attributes can be represented as a five-tuple in the form (name, x−position, y−position, width, height, text, image). Here the first element of the tuple describes the name of the component. The next four elements of the tuple describe the location of the top left point for the bounding box of the component, and the height and width attributes describe the size of the bounding box. The text attribute corresponds to text displayed (if any) by the component, and the image attribute represents an image (if any) of the component with bounds adhering to the position attributes.

All of the developers demonstrated that they always preprocess the icon designs before implementing, due to the gap to designers. D9 said, “We are all fulfilling of our goals, but in different directions.” At the core of it, designers are focused on the graphical/visual representations of icons, whereas the developers take care of the functionality of icons, such as efficient rendering. However, this gap points to a more prominent problem for icon implementation in practice, as D10 showcased an icon design in the design artifact (as shown in Figure 1) and said:

In most big IT or software companies, the visual designers design the UI with professional tools like Sketch [93] or Adobe Photoshop [83]. These tools provide build-in curvature pen/pencil tool allow designers to create some complicated components by drawing smooth curves and straight line segments with equal ease in order to sketch...
out the desired shape. Well-designed icons are usually composed of images, curves, lines, shapes, etc. For example, in Figure 1, the “gift” icon is composed by lines, shapes, and bitmaps. Developers need to spend extra effort to compose all the small pieces into an icon to gain icon rendering efficiency and code accessibility.

The developers also mentioned that finding all of the components of an icon is surprisingly a non-trivial task. This is because that as the designers are only concerned with the visual representation of icon, they may create one icon with many components scattered all over the design artifact. For example, the bitmap (“knot-bow”) is a component of the “gift” icon but not in the same hierarchy of other components in Figure 1. D9 supported this claim:

Although there is much information (i.e., name, attribute, hierarchical relationship, etc.) in the design artifact, that information does not align well with that required in the icon composition and implementation. In practice, I potentially consider three characteristics to compose an icon. First, the rendering of the components of icon should be close/overlap to each other. Second, some designers may organize the resources of an icon with some layout hierarchies to group them, for example “group6” in Figure 1. Third, some designers may use the same format to name icon.
components, for example “gift-piece-x.” Therefore, it takes time to search all the components with reference to their rendering, hierarchy, and attribute.

Due to the gap between designers and developers, developers can hardly implement icons from design artifacts directly. Given a design artifact, developers are required to manually compose the icon image by searching all related components based on their attributes, hierarchies, and renderings, adding to the complexity and effort required for icon implementation in practice.

3.2 Research Question 2: Do Developers Implement Icons in Fonts or Images?

By summarizing the approaches, we collected four ways of rendering icons—that is, image tag `<img>` or `<svg>`, icon tag `<i>`, CSS background image, and custom tag `<SvgIcon>` as shown later in Table 4. One-third of our developers listed all approaches, and 80% of developers knew the way to use images and fonts. Two developers never heard of or used the fonts to render icons. D2 said:

Making front-end development is fun, although sometimes it hurts because I do not have adequate learning experience. There are few front-end courses in universities, and these courses usually contain relatively simple knowledge, such as what is `<div>` block, how to connect HTML and CSS together, etc. They do not teach the usage of font, especially they do not distinguish the difference between fonts and images in rendering icons.

Seventy percent of developers implemented icons as an image when developing front-end codes based on UI design draft files because they found that converting an icon to a font is a complicated and laborious process. For example, D7 mentioned:

To implement the approach of icon font, I first need to upload the image to the existing conversion websites such as icomoon [49] and Fontello [38]. Then, I need to download the generated icon font to my local device. Last but not least, I need to copy the generated CSS code to CSS files. This entire process requires a lot of time and effort, but due to time constraints, the process is not compatible in industry.

One developer, D2 from Alibaba, described how to limit the time in UI development:

Every year, Alibaba has more than 240 events which stores offer special discount, such as Double 11 Global Delight Event, Tmall Thanksgiving Day Event, 1212 Global Discount Event, etc. Due to the high demand for the UI development in the duration of events, we are required to implement UIs in 3 or 4 days.

Developers also considered the trade-off between UI performance and its value. Since the usage of an icon font does not provide business value, it is often a low priority in industry. Even if they knew the benefit of using a font, they would not put effort into doing this. For instance, D9 explained:

Although I know the icon font is better compared to icon image, I will not apply this approach in development. I usually have 3 tasks in a week, such as UI implementation, bug testing, algorithm implementation, etc. I agree that icon font can improve UI rendering performance and provide better user experience. But, the overall functionality will still work without icon font. In contrast, without bug testing, the front-end codes may not work, resulting in significant impact on the company business. And if I do not implement the algorithm, other developers will not be able to apply the API in their development, which will slow down the development speed and delay the product release time.
Another example shows that a potential gap between industry and individual is that 50% of developers mentioned that they use a font to render icons when developing their personal projects, such as a homepage, blog, or tutorial. For example, D2 said:

> When I developed my first personal website, I discovered Font Awesome [39], a font toolkit to render icons by simply adding class description. Since this is my website, I can design freely according to my preferences. To quickly develop my website, I used the font in Font Awesome to implement all the icons in my website. However, it is not applicable in industry. In industry, every icon is well designed according to the company culture and design specifications. Therefore, it is not suitable to apply widely used icon font resources from online platforms. In addition, using online icon fonts involves intellectual property (IP) issues which must be avoided in the industry.

Despite that most of the developers know the benefits of using a font to render icons, few of them implement a font in practice. The icon they used is distinct to the online resources, as it comprises company culture and design guidelines. Therefore, rather than directly using the online resources, developers have to spend extra effort in converting icons to fonts, which is time consuming and laborious.

### 3.3 Research Question 3: Do Developers Write Descriptions for Icons?

All of the developers mentioned that they wrote descriptions/comments in their personal projects, such as assignments and homepage. However, half of the developers did not write descriptions in practice due for the following practical reasons. First, since the readability of code is not a mandatory requirement, many developers did not write well-formatted descriptions for code. For example, the code in the industry cannot be released as open source. D9 said,

> “Since our code can not be released to public, I would not spend too much time on writing comments in code because only a few internal developers would collaborate on my tasks.”

In addition, since updating iteration in the industry is fast, it is not worth it to put too much effort in commenting, especially for icons. For example, D10 said:

> In the year of 2019, our company developed over 1 million UIs. Due to the diversity of UIs, few designs are re-implemented and few code are reviewed. Because of the fast updating iteration and low reusing rate, I did not write well-formatted comment, particularly for the images. I was developing a shopping application which images cover more than half of the UIs. To develop the large amount of images quickly, I prefer using `<img>` tag without any alternative description.

Second, 80% of developers mentioned that writing a well-understood description is a challenging task. It requires developers to understand the intention of the icons, although few developers pay attention to the content of the UIs. For example, D7 explained:

> I agree that the clear descriptions in the code can keep the code readable and “save lives” while unreasonable descriptions “kill lives”. However, it is hard to write a good description. Here is the process of how I write the descriptions: Firstly, I design a comment for every component, image . . . based on its characteristic. Secondly, I rename and simplify the comments according to practical requirement. Thirdly, I check if the comments match the content of UIs or not. Then I repeat this process until the deadline. And obviously, the process is time-consuming and not applicable in the industry.
Despite that the insufficient descriptions in the code may not impede professional developers, it creates a significant cognitive burden for interns and new developers. For example, D3 said:

I am a junior student who came to the company for internship. The first task assigned to me by my leader was to understand the code. However, I found that most of the code is uncommented, which makes it very difficult for me to understand. To understand this part of code, I asked more than 5 developers who participated this project. These uncommented codes negatively influenced my work.

Developers rarely write descriptions for images, especially for icons, because the loose restriction on code readability makes developers care less about code descriptions. Most of developers agree on the difficulty of designing simple, concise, and easy-understood descriptions. The lack of description can adversely affect novice employees and lead to inadequate understanding of the code.

4 ICON CHARACTERISTICS

In this section, we carry out an empirical study of collaborative icon labeling in online icon design sharing websites to understand its characteristics for motivating the required tool support. There are numerous online icon design sharing websites, such as Font Awesome [39] and Google Material Design Icons [43], which provide a comprehensive icon library to assist designers and developers in designing and coding. In these online icon websites, each label matches only one icon design. However, in a real case, one label may have several different designs, revealing the limited diversity of these websites. In this work, we select the Alibaba Iconfont website [3] as our study subject—not only because it has gained significant popularity among in the design community but also because it has become a repository of knowledge with millions of diverse icon designs created by designers. To collect icons and associated labels, we built a web crawler based on the breadth-first search strategy [78]—that is, collecting a queue of URLs from a seed list and putting the queue as the seed in the second stage while iterating. The crawling process continued from December 12, 2019, to July 1, 2020, with a collection of 100k graphical icons.

4.1 Overview

During the process of open coding the categories of icons semantically, we find that one label can be written in different styles. For example, the label “crop” can be written in not only its standard format but also its derivations synonyms like “prune,” “clip,” and “crop-tool.” Moreover, considering that the icon labeling process in Iconfont is informal and icon designs are contributed by thousands of designers with quite diverse technical and linguistic backgrounds, the same concept may labeled in many user-defined terms such as “crop-fill,” “crop-portrait,” and “icon-crop-solid-24px.” The wide presence of forms poses a serious challenge to the icon classification task. For example, the icon can be described to the class of “crop” or “clip,” which makes sense in both classes.

To address the problem, we adopted association rule mining [1] to discover label correlations from label co-occurrences in icons. We leveraged the visual information from the icons and textual information from the labels to group a pairwise correlation of labels. For measuring the visual similarity, we adopted the image similarity score MSE [100] $\text{sim}_{\text{vis}}(x, y)$ to calculate the likelihood if two icons are the same. For measuring the textual information, we first trained a word embedding [72] model to convert each label into a vector that encodes its semantic. Then we defined a lexical similarity threshold based on the string edit distance [57] $\text{sim}_{\text{text}}(x, y)$ to check if two labels are similar enough in the form. The labels are grouped as a pairwise correlation if $\text{sim}_{\text{vis}}(x, y) \geq 0.9$ or $\text{sim}_{\text{text}}(x, y) \geq 0.9$. As we wanted to discover the semantics and construct a lexicon of categories,
we found frequent pairs of labels. A pair of labels is frequent if the percentage of how many icons are labeled with this pair of tags compared with all icons is above the minimum support threshold $t_{\text{sup}} \geq 0.001$. Given a frequent pair of labels $\{t_1, t_2\}$, association rule mining generated an association rule $t_1 \Rightarrow t_2$ if the confidence of the rule $t_{\text{conf}} \geq 0.2$. Given the mined association rules, we constructed an undirected graph $G(V, E)$, where the node set $V$ contains the labels appearing in the association rules and the edge set $E$ contains undirected edges $< t_1, t_2 >$ (i.e., pair of label associations) if the two labels have the association $t_1 \Rightarrow t_2$ or $t_2 \Rightarrow t_1$. Note that the graph is undirected because association rules indicate only the correlations between antecedent and consequent. All threshold values were carefully selected through a manual check, considering the balance between the information coverage and overload.

To identify the set of frequently occurring icon label categories, we performed an iterative open coding of the most frequent co-occurring labels (or approximately 9.2\% of the dataset, 542,334 in total) with an existing expert lexicon of categories in books and websites such as Google’s Material icon set [43], IBM’s Design Language of Iconography [48], and Design Pattern Gallery [79]. Two researchers from our team independently coded the categories of these labels, noting any part of the initial vocabulary. Note that both researchers have design experiences in both icons and UI development. After the initial coding, the researchers met and discussed the discrepancies and the set of new label categories until consensus was reached. Semantic icon categories can be seen in Table 1. We observed two distinct characteristics in icons compared to the physical-world objects.

**High cross-class similarity.** Icons of different classes often have similar size, shape, and visual features. The visual differences to distinguish different classes of icons can be subtle—in particular, small widgets are differentiated by small visual cues. For example, the difference between “newspaper” and “file” lies in a text of news at the top/bottom side of “newspaper,” whereas a plus/minus symbol distinguishes “zoom_in”/”zoom_out” from “search.” In addition, direction is an important aspect to distinguish classes. For example, the inclined waves represent “signal,” and the upward waves represent “wifi.” Existing object classification tasks usually deal with physical objects with distinct features across classes, for example, fishes, flowers, hockey, and people in the popular ImageNet dataset [28]. High cross-class similarity affects classification, as the class cannot be easily distinguished.

**Small, transparent, and low contrast.** To make UI unique and stylish in the screen, icons are usually small and partially transparent, such as the last icon in the “minus” class shown in Table 1. The transparent icons in the UIs do not cause vision conflict, and they are less visible when separated from the background context. For example, the first icon in the “text” class in Table 1 is an icon with low color contrast and uses transparency and shadow to stress contrast, whereas the contrast of the object is obvious in the current dataset, and especially apparent in the grayscale format, such as in the MNIST dataset [55].

| Existing icon sharing sites contain a wide presence of forms of labeling. Based on different background knowledge, designers use different same-meaning labels to annotate the same icon. Such limitation not only confirms our finding of difficulty of commenting in Section 3.3 but also hinders the potential challenge in the classification task. Therefore, a data mining approach capturing visual and textual information is applied to construct a lexicon in icons. By observing the lexicon, we find two distinct characteristics of icons different from the existing physical object orientated dataset. |

5 APPROACH

Based on the interview study in Section 3, we summarized three design considerations for our approach: (1) icons are assembled from multiple components in the design artifact, (2) icons
Table 1. The 40 Icon Classes Identified Through an Iterative Open Coding of 100k Icons from Iconfont [3]

| CLASS      | ASSOCIATED LABEL                                      | EXAMPLES | NUMBER |
|------------|-------------------------------------------------------|----------|--------|
| add        | plus, addition, increase, expand, create             |          | 357    |
| calendar   | date, event, time, planning                          |          | 324    |
| camera     | photo, take-photo                                    |          | 355    |
| chat       | chat-bubble, message, request, comment               |          | 372    |
| complete   | finish, confirm, tick, check, ok, done               |          | 432    |
| computer   | laptop, device, computer-response, desktop           |          | 521    |
| crop       | prune, crop-tool, shear, clipper crop-portrait       |          | 436    |
| download   | file-download, save, import, cloud                  |          | 444    |
| edit       | editing, handwriting, pencil, pen, edit-fill, modify |          | 546    |
| emoji      | amoebe, sad, happy, emotion                          |          | 374    |
| envelope   | letter, email, mail, inbox                           |          | 332    |
| exit       | quit, close, switch-off, logout                      |          | 404    |
| flower     | flowers, flower pot, sunflower, valentine-flower     |          | 377    |
| gift       | present, reward, surprise                            |          | 340    |
| house      | home, rent, house-area, house asset, building, mall  |          | 378    |
| left       | return, back, prev, backwards                        |          | 531    |
| like       | thumb-up, heart, vote, hand-like, upvote, dislike,   |          | 386    |
|            | favourite                                            |          |        |
| location   | gps, direction, compass, navigation                  |          | 543    |
| menu       | menu-file, card, menu-fold, menu-line, more, dashboard|        | 351    |
| minus      | remove, minus (with circle), minus-sign              |          | 556    |
| music      | music-note, music-library, musical-instrument        |          | 375    |
| news       | newspaper, info, announcement                        |          | 423    |
| package    | package-up, package-sent, package, personal          |          | 362    |
| pay        | money, wallet, dollar, commerce                      |          | 364    |
| person     | user, avatar, account, customer                      |          | 562    |
| photo      | image, picture, camera                               |          | 481    |
| play       | playicon, broadcast, play voice, play button, play   |          | 498    |
|            | arrow                                                |          |        |
| question   | ask, faq, information, help, info, support           |          | 350    |
| refresh    | reload, sync, reset, recreate                        |          | 321    |
| right      | forward, next, go, arrow-forward                     |          | 412    |
| safe       | safe-box, safety, safety certificate, lock, secure   |          | 476    |
| search     | investigate, search-engine, magnifier, find, glass   |          | 377    |
| send       | send-arrow, paper-plane, message                     |          | 318    |
| settings   | toolbox, gear, preferences, options                 |          | 317    |
| shopping   | cart, shopping-bag, checkout                         |          | 472    |
| signal     | signal-tower, wave, radio, broadcast                 |          | 548    |
| sound      | speaker, sound volume, player                        |          | 415    |
| star       | collection, rate, favourite                         |          | 424    |
| switch     | switch-on/off, switcher, open, close                 |          | 319    |
| text       | word, textbox, font, size                            |          | 446    |
| visibility | visible, show, hide, visibility-off, in-sight        |          | 339    |
| warn       | alarm, warning, error, report, alert                 |          | 369    |
| wifi       | wi-fi, wireless, network, signal                     |          | 429    |
| zoom-in    | fullscreen, expand, adjust, magnifier                |          | 384    |
Fig. 2. The approach of our tool, Auto-Icon+. Given a design artifact as input, we first compose the components into icons by clustering (e.g., the “eye,” “mouth,” and “face” components are composed into a smiling face icon). We then apply three major functions, involving font conversion, prediction, and color detection, to generate the output code for the icon, which consists of a font file (.ttf) and a descriptive tag (e.g., “icon-emoji” and “blue”).

implemented as fonts can expedite rendering, and (3) adding descriptions to icons can increase readability and accessibility of code.

To solve (1), we proposed a machine learning technique to cluster the scattered components in the design artifact through a heuristic correlation coefficient and then composed the clusters of components into icons (Section 5.1). To solve (2), we proposed an automated conversion technique, taking an icon as the input, and outputting a vector graphics font (Section 5.2). To address (3), we were inspired from the findings in our study on icon characteristics in Section 4 to develop a deep learning model to automatically assign the classes of icons to reduce the effort of manually designing the description of icons (Section 5.3). Additionally, we applied a primary color detection method based on computer vision to keep track of the primary color of the icon to support a more detailed description in code (Section 5.4). The overview of our approach is shown in Figure 2.

5.1 Icon Composition

The most common way to group related pieces is through clustering. Normally, the assumed number of clusters may be unreliable since the number of icons among the design artifacts is unknown and thus the top-down partitioning methods (K-Means [60], Expectation-Maximization [27], etc.) will not applicable. To provide clustering without requiring the knowledge of clusters, we adopted a bottom-up approach Hierarchical Agglomerative Clustering (HAC) [77]. The details of HAC are shown in Algorithm 1, where each component in the design artifact is treated as a singleton cluster to start with, and then they are successively merged into pairs of clusters until all components have merged into one single large cluster. The main parameters in this algorithm are the metric used to compute the correlation value of components, which determines the pair of clusters to be merged at each step. According to our observations in Section 3.1, we defined a heuristic correlation metric that takes into account the component’s attribute, hierarchy, and rendering:

$$\text{Correlation}(x, y) = \alpha \times \text{ATTR}(x, y) + \beta \times \text{HRCHY}(x, y) + \gamma \times \text{IOU}(x, y),$$

(1)

where $\text{ATTR}(x, y)$ measures the attribute type of two components $x, y$—that is, if they have same type (image, curve, shape, etc.), it assigns the value to 1 else 0. $\text{HRCHY}(x, y)$ measures the
Fig. 3. The HAC algorithm to compose components into icons in the design artifacts, which repeatedly select and merge pairs of clusters based on a heuristic correlation metric, until a single all-inclusive cluster (UI). By empirically setting a correlation threshold to 0.6 and distinguishing the icon cluster by its feature as discussed in Section 4.1 (small, transparent, and of low contrast), the blue cluster is determined as an icon.

**ALGORITHM 1:** Hierarchical Agglomerative Clustering (HAC)

| Line | Description |
|------|-------------|
| 1    | **Input:** UI design artifact $X$ with $n$ elements $\{x_1, x_2, x_3, \ldots, x_n\}$ |
| 2    | **Output:** Hierarchical Dendrogram $HD$ |
| 3    | construct $n \times n$ matrix $HD$ with correlation metric $cor(i, j)$ between the elements; |
| 4    | **while** $\text{len}(X) > 1$ **do** |
| 5    | | Select the pair $(x_i, x_j)$ with the largest correlation value, such as $x_i, x_j \in X$; |
| 6    | | Merge the pair $(x_i, x_j)$ into a new cluster $x_{merge} = x_i \cup x_j$, let $x_i, x_j$ be the sub clusters of cluster $x_{merge}$; |
| 7    | | Update $X \leftarrow X \cup \{x_{merge}\} - \{x_i, x_j\}$; |
| 8    | | **foreach** $x \in X$ **do** |
| 9    | | Update the matrix $HD$ with correlation metric $cor(x, x_{merge})$; |
| 10   | **end** |
| 11   | **return** $HD$ |

hierarchy between components, and the value is assigned to 1 if they are under the same group and 0 otherwise. $\text{IOU}(x, y)$ measures the overlap between components, taking a value between 0 and 1. In addition, $\alpha, \beta, \gamma$ are the user-defined weights for each of the measurements. The higher the correlation value, the more likely the cluster can composed to an icon.

The agglomeration of clusters results in a tree-like structure called the *dendrogram* as shown in Figure 3. The value is highest at the lowest level of the dendrogram, and it decreases as the clusters merge into the final single cluster. By cutting the dendrogram at an empirical setting of the value threshold (e.g., 0.6), we may retrieve several clusters, such as the icon (blue cluster), text (red cluster), and background (green cluster) in Figure 3. To further identify if the aggregated cluster is an icon (blue cluster), we followed the feature of the icon discussed in Section 4.1, in which the icon should be small and of low contrast.

### 5.2 Font Conversion from an Icon

Unlike converting a font to an image, transcribing an image to a font, which is also known as the image tracing problem, is a difficult task. In this work, we adopted the state-of-the-art Potrace [90]
in Figure 2(A). We first applied a preprocessing method for converting color to a binary (i.e., black and white) image by setting a threshold to control the bit of each pixel after calculating the average value in three channels \((R + G + B)/3\). We regarded the pixel white if the average value was larger than 128, whereas we regarded the pixel black if the value was equal to or smaller than 128. Then, we detected the edge in the black-white image. An edge was defined to be a border between a white pixel and a black pixel, which indicates which pixels from the original image constitute the borders of a region. Note that the edge was assigned a direction so that when moving from the first endpoint to the second, the black pixel was on the left (as shown in Figure 2(A) edge detection). This process was repeated until we reached the starting point, at which point we found a closed path that enclosed a black region. Once the border was found, we approximated/optimized the border with a polygon to figure out which border pixels were possible to connect with a straight line such that the line passed through all border pixels between its endpoints. To detect the optimal polygon, we computed a penalty value to measure the average distance from the edge to the pixels it approximated. The polygon with the smallest penalty (equivalent to the polygon with the fewest pixels) was the optimal one. Finally, we used a cubic curve defined by four control points (also known as the Bezier curve [89]) to smooth the corners. The first and fourth control points (i.e., midpoints of the edges of the polygon) gave the locations of the two endpoints of the curve, whereas the second and third (i.e., chosen on the polygon edges through the endpoints) indicated the direction and magnitude of the derivative of the curve at each endpoint.

5.3 Prediction for an Icon

The traditional CNN [53, 55] has shown great potential as a solution for difficult vision problems. MobileNetV2 [88] distills the best practices in convolutional network design into a simple architecture that can serve with competitive performance but keep low parameters and mathematical operations to reduce computational power. The architecture of the network is shown in Figure 2(B).

Instead of using regular convolutional layers widely used in traditional CNN architectures to capture essential information from images but are expensive to compute, MobileNetV2 adopted a more advanced one—depthwise separable convolutions. Depthwise separable convolution combined a \(3 \times 3\) convolution layer and two \(1 \times 1\) convolution layers. The \(1 \times 1\) convolution layer (also named the pointwise convolution layer) was used to combine the filter values into new features, whereas the \(3 \times 3\) convolution (also called the depthwise convolution layer) was used to filter the input feature map. Inspired from the dimension augmentation in the work of Lin et al. [61], MobileNetV2 used a \(1 \times 1\) pointwise convolution layer to expand the number of channels in the input feature map. Then it used a \(3 \times 3\) depthwise convolution layer to filter the input feature map and a \(1 \times 1\) convolution layer to reduce the number of channels of the feature map. The network borrowed the idea of a residual connection in ResNet [45] to help with the flow of gradients. In addition, batch normalization and an activation layer were added between each depthwise convolution layer and pointwise convolution layer to make the network more stable during training. For detailed implementation, we adopted the stride of 2 in the depthwise convolution layer to downsample the feature map. For the first two activation layers, the network used ReLU6 defined as \(y = \min(\max(0, x), 6)\) because of its robustness in low-precision computation [46], and a linear transformation (also known as the linear bottleneck layer) was applied to the last activation layer to prevent ReLU from destroying features.

5.4 Color Detection of an Icon

Since the conversion between icon and font sacrifices the color identity, we added an attribute to keep track of the primary color of the icon. To that end, we adopted HSV colorspace for color detection. We first removed the fourth alpha channel as transparent and made a conversion from
RGB color to HSV colorspace. Each RGB color has a range of HSV value. The lower range is the minimum shade of the color that will be detected, and the upper range is the maximum shade. For example, blue is in the range of \(\langle 100, 43, 46 \rangle\)–\(\langle 124, 255, 255 \rangle\). Then, we created a mask for each color (black, blue, cyan, green, lime, magenta, red, white) as shown in Figure 2(C). The mask is the area in which the HSV value on pixels matches the color between the lower range and upper range. Finally, we calculated the area of the mask in each color and the corresponding image occupancy ratio. The color with the maximum ratio was identified as the primary color of the icon (the blue in the example in Figure 2(C)).

6 EXPERIMENTS

In this section, we first set up an experiment to analyze the performance of our tool. Then we conduct a pilot user study to evaluate the usefulness of our tool. Furthermore, we demonstrate its usefulness on a large-scale industrial benchmark. The goal of our experiments is to answer the following research questions, in terms of accuracy, efficiency, and applicability: RQ1: How accurate is our heuristic algorithm in composing icons from design artifacts? RQ2: How accurate is our model in predicting labels for icon images? RQ3: How much do our tools increase the efficiency of UI development? RQ4: What are the developers’ opinions on the usability of our tool?

6.1 Icon Composition

6.1.1 Dataset. To evaluate the performance of our heuristic composition algorithm, we compare our automatically generated icon to manually generated ground truth. To generate the ground truth, we had 1 author and 10 non-author paid annotators independently label the components in the design artifacts (e.g., the component belongs to not-icon or icon\(_x\), and as one artifact may contains multiple icons, we apply \(x\) to annotate icon separately). Each design artifact was labeled by three different annotators, and the ground truth was generated until an agreement was reached. All annotators have previous experience designing and implementing icons. In total, the annotators labeled 1,012 real-world design artifacts with 9,883 icons.

6.1.2 Baselines and Metrics. Since the number of icons in the design artifact is unknown, we set up two widely used clustering algorithms that do not need to pre-set the number of clusters in advance as baselines: Mean-Shift clustering \([24]\) and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) \([31]\). To further gauge the advantages of our heuristic icon composition algorithm, we compare the icon produced by our correlation metric to each metric individually:

**Mean-Shift:** Mean-Shift is a centroid-based clustering algorithm to detect the close components in the design artifact. It works by updating the candidates for the center points as the mean of the points within the sliding window.

**DBSCAN:** DBSCAN is an improvement over Mean-Shift clustering, as it involves a transitivity based chaining approach to determine whether components are located in a particular cluster, to separate clusters of noise and outliers.

**ATTR only:** ATTR only composes the icon based on the component’s attribute in the artifacts—for example, the curve components are merged together as an icon.

**HRCHY only:** As designers may manage all resources of the icon under a layout folder, we rely on this hierarchical structure information to assemble the icon.

**IOU only:** The rendering of components in the icon are likely intersecting and overlapping to each other, and therefore we compose the icon based on measuring the IOU of rendering.
Table 2. Performance Comparison for Icon Composition

| Method       | Mean-Shift | DBSCAN | ATTR only | HRCHY only | IOU only | Ours  |
|--------------|------------|--------|-----------|------------|----------|-------|
| Precision    | 40.16%     | 55.54% | 27.10%    | 3.23%      | 45.40%   | 76.50%|
| Recall       | 51.91%     | 73.34% | 35.20%    | 6.25%      | 62.51%   | 81.25%|
| F1-score     | 45.28%     | 63.21% | 30.62%    | 4.26%      | 52.59%   | 78.80%|

To evaluate the performance, we set up three evaluation metrics: precision, recall, and F1-score. Precision is the proportion of icons that are correctly composed among all icons in the design artifacts. Recall is the proportion of icons that are correctly composed among all composed components (e.g., icon and not-icon). F1-score is the harmonic mean of precision and recall, which combine both of the preceding two metrics.

6.1.3 Results. Table 2 shows the performance of all methods. The centroid-based clustering algorithm can only achieve 45.28% and 63.21% F1-score for Mean-Shift and DBSCAN, respectively. The issues with these baselines are that they are designed for large and dense data, such as natural disasters and stations. However, different from those data, the number of components in design artifacts is relatively small. The performance of our algorithm is much better than that of other correlation baselines—for example, 31.1%, 18.74%, and 26.21 boost in precision, recall, and F1-score compared with the best correlation baseline (IOU only). Relying only on hierarchical structure information (HRCHY only) yields poor performance in all metrics, 3.23%, 6.25%, and 4.26%, respectively, compared to the ground truth. This is because many designers follow their own design styles, which highly differ from each other, and some of their designs miss the rigid layout information. ATTR only yields an average F1-score of 30.62%, which is better than the F1-score of HRCHY only, indicating that the attribute of components provides better information for icon composition, as designers tend to use consistent attributes for icon designs. IOU only yields the best performance among all baselines with an average F1-score to 52.59%, indicating that the icons are normally composed by several intersecting components. Figure 4 shows some results of icon composition by our heuristic algorithm. It shows that our algorithm can help cluster the components with different attributes (e.g., bitmaps and path for the first icon), different hierarchies (e.g., “Oval” component is not in the layout for the second icon), and no intersection (e.g., the components of the icon have spacing for the third icon).

To identify the common causes of errors, we further manually checked the wrong composition cases in the test dataset. According to our observation, we found two main reasons. First, the components of icons are widely spaced, such as three separated waves for “wifi,” three horizontal lines for “menu,” and several lines from low to high for “signal.” In addition, designers may depict some effects around the icon, such as the ripple effect on a hand for “tap” and the shiny effect on a bulb for “light.” The wide space between components decreases the correlation value of IOU, resulting in poor clustering. Second, some designers place the icon and text in the same hierarchy to depict an image button—for example, a folder consisting of icon components, text components, button border, and so on, causing our algorithm to aggregate an image button rather than an icon.

6.2 Icon Prediction

6.2.1 Dataset. We leveraged the categorization during the creation of the semantic vocabulary (in Table 1), and corresponding icons and attached labels as the training data. The foundation of the deep learning model is the big data, so we only selected categories with frequency larger than 300 for training the model. Therefore, there were 100 categories left with the number of icons ranging from 311 to 589. Given all of the data assigned to each label, we randomly split these 41k
Auto-Icon+

Fig. 4. The results of our icon composition algorithm. The top represents the design artifact, and the bottom represents the clustering steps.

Table 3. Label Classification Accuracy and Time Estimation in Different Methods

| Method           | Histo + SVM | Histo + DT | SIFT + SVM | SIFT + DT | ResNet-50 | VGG-16 | MobileNetV2 (RGBA) | MobileNetV2 (RGB) |
|------------------|-------------|------------|------------|-----------|-----------|--------|-------------------|-------------------|
| Accuracy         | 0.5657      | 0.3267     | 0.5806     | 0.4686    | **0.8839**| 0.8764 | 0.8348            | 0.8772            |
| Time (ms)        | 0.103       | 0.152      | 1.702      | 1.941     | 26.535    | 27.282 | 17.567            | **17.485**        |

icons into train/validation/test datasets with a ratio of 8:1:1 (33K:4K:4k). To avoid the bias of “seen samples” across training, validation, and testing, we performed fivefold cross validation.

6.2.2 Baselines. We set up several basic machine learning baselines including the feature extraction (e.g., color histogram [99], scale-invariant feature transform [66]) with machine learning classifiers (e.g., decision tree [85], SVM [25]). Apart from these conventional machine learning based baselines, we also set up several derivations of state-of-the-art deep learning models as baselines to test the importance of different inputs of our approach including backbones (ResNet [45], VGG [91], MobileNet [88]) and different input channels (RGB, RGBA). The training and testing configurations for these baselines were the same.

6.2.3 Results. As we trained a classifier to predict label for the icon, we adopted accuracy as the evaluation metric for the model, as illustrated in Table 3. The traditional machine learning method based on the human-crafted features can only achieve about 0.6 average accuracy. Deep learning models perform much better than the best old-fashioned methods (i.e., with 0.3033, 0.29712, and 0.2958 increase for ResNet-50, VGG-16, and MobileNetV2, respectively). Although the ResNet model performs the best in the icon classification task, it requires a relatively long time for prediction (26.535 ms per icon), which strongly violates the performance of UI rendering (16 ms) as we aim to deploy into an online platform (Imgcook). In contrast, our model MobileNet is nearly as
Table 4. Examples of Development for Icons in Experimental (E) and Control (C) Groups

| E                  | C1                                      | C2                                      | C3                                      |
|--------------------|-----------------------------------------|-----------------------------------------|-----------------------------------------|
| <i class="icon-left red"></i> | <img src="8E431911-61BB-4A19-8C01.svg"/> | <img src="next-icon-design.svg" alt="next" width="100%"/> | <div style="background-image: url(8E431911-61BB-4A19-8C01.svg);"></div> |
| E                  | <i class="icon-information white"></i> | <svg class="icon" aria-hidden="true">  | <svg>                                    |
|                    | <use xlink:href="icon-information"/></use> | </svg>                                  | </svg>                                  |
|                    | <a style="background-image: url(q&a.svg); width: 100%, height:100%;" class="help-icon"> </a> |                                       |                                        |
|                    | <a class="svg-icon" id="#"></svgIcon name="white"></a> |                                      |                                         |

accurate as ResNet with a performance lag of 0.67% while being 34.1% faster. In addition, we find that the increase of a fourth alpha channel (RGBA) decreases the accuracy from 0.8772 to 0.8348 for two main reasons. First, the result shows that the model with RGB input has a loss value of 0.7844 at epoch 200, which is better than the model with RGBA (0.9231). This is because the supplemented channel greatly increases the parameters of the model, which leads to a decline in the ability of gradient training at the same epochs. Second, based on the principle of optics, the fourth alpha channel does not reflect the morphological characteristics of the image. It is used to reduce information of the other three channels by adjusting their color/degree, causing less information to be captured through the training process.

6.3 User Study

6.3.1 Procedures for the User Study. Ten developers (generators), all proficient in UI development and have at least 1 year of experience, were recruited for this study. We randomly selected five icons from the real-life UI designs and asked each generator to develop them. To guarantee the generators can objectively develop the icons, we asked whether they had prior knowledge of the icons (e.g., development experience, design experience). The time of the development was recorded. To be fair, generators did not know that we were recording the time, as the time pressure may have affected their development (in quality, speed, etc.) [8, 68]. We set the manual development as the control group. Then, we also asked them to develop five other icons with the help of our tool that not only automatically convert the image to font but also provide the description (predicted label and color). We labeled this the experimental group. The detailed developments of two groups for icons are shown in Table 4.

We then recruited another 10 developers (evaluators), and each of them was assigned the developments from two control groups and one experimental group. Note that they did not know which one was from the experimental or control group, and for each icon, we randomly shuffled the order of candidates to avoid potential bias. Given each development, they individually marked it as readable or not in a 5-point Likert scale (1: not readable at all and 5: strongly readable). To
evaluate the performance of usability, we also asked evaluators to rate how likely they would be to use the development in practice (Acceptable). The measurement was also in a 5-point scale.

6.3.2 Results. The box plot in Figure 5 shows that the time spent on the development of icons in the experimental group is much shorter than that in the control group (with an average of 6.05 seconds versus 17.39 seconds, saving 65.2% of the time). That is the biggest strength of our tool—that is, developers can quickly develop an icon by providing descriptions and a font pattern. On average, the overall readability ratings for the experiment group is 4.16, which are significantly higher (48.5%) than those of the control group (2.8) in Figure 5. Most developers admit that our tool can provide more acceptable results for them. In other words, 94% (4.7/5.0) of developers hope to develop the icons with the help of our tool in their real development environment compared to 3.96 in the control group. To understand the significance of the differences, we carry out the Mann-Whitney U test [32] (specifically designed for small samples) on the readability and acceptability ratings between the experiment and the control group, respectively. The test results suggest that our tool significantly outperforms the baseline in terms of these metrics with \( p < 0.01 \) or \( p < 0.05 \).

For some icons, the developer gave a very low acceptability score to the labels. According to our observation and further survey, we summarize two reasons accounting for those bad cases. First, albeit the good performance of our model, we still make wrong predictions for serendipitous icons. Based on the context of icons, the same icon can have different meaning. For example, the icon in Figure 6 represents the meaning of "information" in the common case, but considering the text on the right, the meaning of the icon should be "glossary"/"dictionary." Second, developers admit the usefulness of converting images to fonts for providing faster rendering speed. However, they also point out the limitation of replacing images with fonts. Fonts are not fully compatible in all browsers and devices. One developer mentioned that they need to make sure that the development works on old devices, in which they usually need to give up the latest efficient methods, such as Iconfont.

6.4 Industrial Usage

We cooperate with *Imgcook* platform [4] developed by Alibaba, an intelligent tool to automatically generate front-end codes from UI design files. *Imgcook* has attracted a lot of attention in the community, which has a large user base (15k) and generates more than 40k UIs. *Auto-Icon* is integrated with the internal automated code generation process and is triggered whenever the design files contain an icon.

To evaluate the usability of our tool, we set up a code review metric for measuring the code modification for icons. Note that the code modification contains multiple contents, such as text and button, and we only measure the modification if the object is an icon to reduce the potential bias. We adopt a case-insensitive BLEU (BiLingual Evaluation Understudy) [82] as the metric to evaluate the preservation of code. BLEU is an automatic evaluation metric widely used in code difference studies. It calculates the similarity of machine-generated code and human-modified reference code (i.e., ground truth) as

\[
BLEU = BP \times \exp \left( \sum_{n=1}^{N} w_n \log p_n \right),
\]

where \( p_n \) denotes the precision (i.e., the ratio of length \( n \) token sequences generated by our method that are also present in the ground truth); \( w_n \) denotes the weight of different length of n-gram summing to 1; and \( BP \) is
1 if \( c > r \), and otherwise \( e^{(1-r/c)} \), where \( c \) is the length of machine-generated sequences and \( r \) is the length of ground truth. A high BLEU score indicates less modification in the code review.

We run the experiment in Imgcook with 6,844 icons in 2,031 UIs from August 20, 2020, to September 20, 2020. Among all of the testing UI developments, the generated code for icons reaches 84% BLEU score, which means that most of the code is used directly without any modification. It demonstrates the high usability of Auto-Icon+ in practice. Based on the inspection results, we categorize the modification into four categories. Two reasons are discussed in Section 6.3.2, in terms of wrong prediction and compatibility concern. There are another two modifications mainly due to industrial practice. First, to maintain the consistency of a company’s coding experience, some developers modify to a prescribed naming/rendering method—for example, packing the icon of “icon-camera” to a <Icon-Camera> tag. Second, UI dynamically changes in practice. Once an element in the UI is changed, the attribute of icon may change, such as color and font size.

Overall, our method achieves a 78.8% F1-score in the icon composition from design artifacts (RQ1) and 87.7% accuracy in the label prediction for icons (RQ2). In the survey of 10 developers, we improve the efficiency of developing time and code readability by 65.2% and 48.5%, respectively (RQ3). The majority (4.7/5.0) of the interviewed developers acknowledge the usability of the generated code for icons by our method, and it is further confirmed in the practice of Imgcook with an 84% BLEU score (RQ4).

7 DISCUSSION

On developers. Due to the conceptual gap between designers and developers, there is a transition barrier from design artifact to implementation. Developers require extra effort to compose design components into icons for optimizing the network traffic and caching. To bridge this transition gap, our work proposes a heuristic machine learning technique to cluster components in the design artifacts to automatically aggregate the scattered icon components into clusters, helping to ease the burden of manual search for developers. The clusters can be further applied to various downstream tasks such as icon implementation, UI testing [76], and CSS clipping [54]. Icon implementation is a challenging and time-consuming task, even for professional developers. On the one hand, UI developers must enhance performance. Poor development has an adverse effect on the performance of the site. The performance issues comprise a multitude of factors like rendering speed, as well as reusability and flow of the code. On the other hand, UI developers must write a clean, high-quality code that can be easily understood and maintained. Inspired by the high performance of font rendering, our work designs an automated method to convert icons to fonts using computer vision techniques to trace the edge of the icon and using a graphic algorithm to optimize the edge. In addition, compared with the missing descriptions in the development or brainstorming suitable names, which is limited to several developers in the physical world, our deep learning and computer vision technique based method can quickly identify the label and color of the icon. Our method, once made accessible to developers, can help developers achieve efficient icon coding.

On the generalization of our method. We report and analyze the performance of our CNN-based model for predicting icon labels in Section 6.2.3. One limitation with our model is that we currently only test our model on 100 labels with enough corresponding icons. With the cooperation of the Imgcook platform, the icons in the UI images are a gold resource. First, the icons are relatively unique; otherwise, developers can reuse the online resources directly. These unique icons can significantly increase the amount of data, consequently improving the accuracy of our model. Second, developers may modify the description to a serendipitous label that can augment the labels and generalize a broader range of icon descriptions. Due to the time limit, we only collect a
small amount of icons from Imgcook. However, we have seen some interesting icons that do not exist on online sharing platforms, and they may improve the generalization of our method.

Area of improvements. Currently, we only predict the label based on the icon itself. As discussed in Section 6.3.2, the meaning of the icon varies in different contexts. To address this problem, we can consider the entire UI, capturing all related information to make the final prediction. Developers praise the idea of adding descriptions to the code, which is a tedious task for them. They wonder whether our model can extend to other elements. One developer wants to support button descriptions for buttons as he finds many buttons do not have descriptive texts to explain its intention, resulting in a bad user experience. We believe our model could help developers in this case, as it will not be difficult to extend to other elements once we obtain enough data for the training. Moreover, developers envision the high potential in being able to add icon size descriptions, as one of the biggest strengths of the font is lossless scalability. To that end, we can measure the height of the icon and map it to the corresponding font size.

8 CONCLUSION

In this article, we present a novel approach, Auto-Icon+, that can provide developers with intelligent support to reduce the development time of icon design in the UI. Our approach consists of two integral parts: a heuristic machine learning method for icon composition, and a deep learning and computer vision method for icon implementation. Our work possesses several distinctive advantages. First, to bridge the conceptual gap between designers and developers around the icon, we propose a heuristic-based clustering method to compose the scattered icon pieces in the design artifact into an icon. Second, we develop an automated image conversion method to turn an icon into a font for faster icon rendering. Third, to assist developers with better code accessibility, we adopt a deep learning model to automatically predict the descriptive label that conveys the semantics of the icon. Fourth, based on the colorspace of the image, we detect the primary color of the icon to provide developers with more knowledge about the icon. Our method is incorporated into an existing automated code generation platform to extend them beyond effective and descriptive coding.

ACKNOWLEDGMENTS

We appreciate Yanfang Chang, Zixiao Zhao, and Jiawen Huang for designing and conducting a survey experiment, and the participants from the Imgcook team of the Alibaba participants from Imgcook team in Alibaba Group for taking surveys.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast algorithms for mining association rules. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB ’94), Vol. 1215. 487–499.
[2] Airbnb. 2021. Sketching Interfaces: Generating Code from Low Fidelity Wireframes. Retrieved May 6, 2022 from https://airbnb.design/sketching-interfaces/.
[3] Alibaba. 2020. Iconfont+. Retrieved September 28, 2020 from https://www.iconfont.cn/.
[4] Alibaba. 2020. Imgcook. Retrieved September 29, 2020 from https://www.imgcook.com/.
[5] Android. 2019. Navigate a Systrace Report. Retrieved September 1, 2020 from https://developer.android.com/studio/profile/systrace/navigate-report.
[6] Gary Ang and Ee Peng Lim. 2021. Learning network-based multi-modal mobile user interface embeddings. In Proceedings of the 26th International Conference on Intelligent User Interfaces. 366–376.
[7] Apple. 2021. Xcode Overview—Apple Developer. Retrieved May 6, 2022 from https://developer.apple.com/xcode/.
[8] Robert D. Austin. 2001. The effects of time pressure on quality in software development: An agency model. Information Systems Research 12, 2 (2001), 195–207.
[9] Michel Beaudouin-Lafon. 2004. Designing interaction, not interfaces. In Proceedings of the Working Conference on Advanced Visual Interfaces. 15–22.
[10] Tony Beltramelli. 2018. Pix2code: Generating code from a graphical user interface screenshot. In Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems. 1–6.

[11] Alison Black. 2017. Icons as carriers of information. In Information Design: Research and Practice. Routledge, 315–329.

[12] Sara Bunian, Kai Li, Chaima Jemmali, Casper Hartevedt, Yun Fu, and Magdy 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.

[13] Margaret Butler. 2010. Android: Changing the mobile landscape. IEEE Pervasive Computing 10, 1 (2010), 4–7.

[14] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. 2017. Investigating the energy impact of Android smells. In Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER’17). IEEE, Los Alamitos, CA, 115–126.

[15] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From lost to found: Discover missing UI design semantics through recovering missing tags. Proceedings of the ACM on Human-Computer Interaction 4, CSCW2 (2020), 1–22.

[16] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinhui 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.

[17] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation. In Proceedings of the 40th International Conference on Software Engineering, 665–676.

[18] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinhui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile GUI components by deep learning. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE’20). IEEE, Los Alamitos, CA, 322–334.

[19] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, and Liming Zhu. 2020. Object detection for graphical user interface: Old fashioned or deep learning or a combination? arXiv preprint arXiv:2008.05132 (2020).

[20] Qiuyuan Chen, Chunyang Chen, Safwat Hassan, Zhenchang Xing, Xin Xia, and Ahmed E. Hassan. 2021. How should I improve the UI of my app? A study of user preferences of popular apps in the Google Play. ACM Transactions on Software Engineering and Methodology 30, 3 (2021), 1–38.

[21] 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 Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE’19). IEEE, Los Alamitos, CA, 596–607.

[22] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. 2019. GUI-squatting attack: Automated generation of Android phishing apps. IEEE Transactions on Dependable and Secure Computing 18, 6 (2019), 2551–2568.

[23] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. 2018. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. Computers & Security 73 (2018), 326–344.

[24] Yizong Cheng. 1995. Mean shift, mode seeking, and clustering. IEEE Transactions on Pattern Analysis and Machine Intelligence 17, 8 (1995), 790–799.

[25] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. Machine Learning 20, 3 (1995), 273–297.

[26] Tobias Dehling, Fangjian Gao, Stephan Schneider, and Ali Sunyaev. 2015. Exploring the far side of mobile health: Information security and privacy of mobile health apps on iOS and Android. JMIR mHealth and uHealth 3, 1 (2015), e8.

[27] Arthur P. Dempster, Nan M. Laird, and Donald B. Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society: Series B (Methodological) 39, 1 (1977), 1–22.

[28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, Los Alamitos, CA, 248–255.

[29] Android Developers. 2020. Inspect GPU Rendering Speed and Overdraw. Retrieved September 28, 2020 from https://developer.android.com/topic/performance/rendering/inspect-gpu-rendering.

[30] Android Developers. 2022. Slow rendering. Retrieved June 16, 2022 from https://developer.android.com/topic/performance/vitals/render.

[31] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD’96). Vol. 96. 226–231.

[32] Michael F. Fay and Michael A. Proschan. 2010. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. Statistics Surveys 4 (2010), 1.
Auto-Icon+

36:23

[33] Ruitao Feng, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, and Shang-Wei Lin. 2019. MobiDroid: A performance-sensitive malware detection system on mobile platform. In Proceedings of the 2019 24th International Conference on Engineering of Complex Computer Systems (ICEeCS’19). IEEE, Los Alamitos, CA, 61–70.

[34] Sidong Feng and Chunyang Chen. 2021. GIFdroid: Automated replay of visual bug reports for Android apps. arXiv preprint arXiv:2112.04128 (2021).

[35] Sidong Feng, Chunyang Chen, and Zhenchang Xing. 2022. Gallery DC: Auto-created GUI component gallery for design search and knowledge discovery. arXiv preprint arXiv:2204.06700 (2022).

[36] Sidong Feng, Suuyu Ma, Jinzhong Yu, Chunyang Chen, Tingting Zhou, and Yankun Zhen. 2021. Auto-Icon: An automated code generation tool for icon designs assisting in UI development. In Proceedings of the 26th International Conference on Intelligent User Interfaces. 59–69.

[37] FluidUI. 2021. FluidUI.com—Create Web and Mobile Prototypes in Minutes. Retrieved May 6, 2022 from https://www.fluidui.com/.

[38] Fonticons. 2020. Font Awesome. Retrieved September 28, 2020 from https://fontawesome.com/.

[39] Fontello. 2020. Icon Fonts Generator. Retrieved September 7, 2020 from http://fontello.com/.

[40] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason Hong, and Norman Sadeh. 2013. Why people hate your app: Making sense of user feedback in a mobile app store. In Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 1276–1284.

[41] Yi Gao, Yang Luo, Daqing Chen, Haocheng Huang, Wei Dong, Mingyuan Xia, Xue Liu, and Jiajun Bu. 2017. Every pixel counts: Fine-grained UI rendering analysis for mobile applications. In Proceedings of the IEEE INFOCOM Conference on Computer Communications (INFOCOM’17). IEEE, Los Alamitos, CA, 1–9.

[42] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. 2016. Mining test repositories for automatic detection of UI performance regressions in Android apps. In Proceedings of the 13th International Conference on Mining Software Repositories. 13–24.

[43] Google. 2020. Material Icons. Retrieved September 9, 2020 from https://material.io/resources/icons/.

[44] Chris Harrison, Gary Hsieh, Karl D. D. Willis, Jodi Forlizzi, and Scott E. Hudson. 2011. Kineticons: Using iconographic motion in graphical user interface design. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 1999–2008.

[45] 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.

[46] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017).

[47] Gang Huang, Mengwei Xu, Felix Xiao Zhu Lin, Yunxin Liu, Yun Ma, Saumay Pushp, and Xuanzhe Liu. 2017. Shuffle-Dog: Characterizing and adapting user-perceived latency of Android apps. IEEE Transactions on Mobile Computing 16, 10 (2017), 2913–2926.

[48] IBM. 2020. UI Icons: Design Language. Retrieved September 9, 2020 from https://www.ibm.com/design/language/iconography/.

[49] IcoMoon. 2020. Font Awesome. Retrieved September 7, 2020 from https://icomoon.io/.

[50] Mohammad Nazrul Islam. 2015. Exploring the intuitiveness of iconic, textual and icon with texts signs for designing user-intuitive web interfaces. In Proceedings of the 2015 18th International Conference on Computer and Information Technology (ICCCIT’15). IEEE, Los Alamitos, CA, 450–455.

[51] Bridgett A. King and Norman E. Youngblood. 2016. E-government in Alabama: An analysis of county voting and election website content, usability, accessibility, and mobilereadiness. Government Information Quarterly 33, 4 (2016), 715–726.

[52] Charalampos Koutsourelakis and Konstantinos Chorianopoulos. 2010. Icons in mobile phones: Comprehensibility differences between older and younger users. Information Design Journal 18, 1 (2010), 22–35.

[53] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems. 1097–1105.

[54] Rob Larsen. 2018. Mastering SVG: Ace Web Animations, Visualizations, and Vector Graphics with HTML, CSS, and JavaScript. Packt Publishing Ltd.

[55] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE 86, 11 (1998), 2278–2324.

[56] Chunghi Lee, Sanghoon Kim, Dongyun Han, Hongjun Yang, Young-Woo Park, Bum Chul Kwon, and Sungahn Ko. 2020. GUIComp: A GUI design assistant with real-time, multi-faceted feedback. In Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. 1–13.

[57] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10 (1966), 707–710.
[58] Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A. Myers. 2021. Screen2Vec: Semantic embedding of GUI screens and GUI components. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. 1–15.

[59] Wenjie Li, Yanyan Jiang, Chang Xu, Yepang Liu, Xiaoking Ma, and Jian Lü. 2019. Characterizing and detecting inefficient image displaying issues in Android apps. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER’19). IEEE, Los Alamitos, CA, 355–365.

[60] Aristidis Likas, Nikos Vlassis, and Jakob J. Verbeek. 2003. The global k-means clustering algorithm. Pattern Recognition 36, 2 (2003), 451–461.

[61] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network in network. arXiv preprint arXiv:1312.4400 (2013).

[62] Mario Linares-Vasquez, Christopher Vendome, Qi Luo, and Denys Poshyvanyk. 2015. How developers detect and fix performance bottlenecks in Android apps. In Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME’15). IEEE, Los Alamitos, CA, 352–361.

[63] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In Proceedings of the 36th International Conference on Software Engineering. 1013–1024.

[64] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes: Spotting UI display issues via visual understanding. In Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE’20). IEEE, Los Alamitos, CA, 398–409.

[65] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Guided bug crunch: Assist manual GUI testing of Android apps via hint moves. arXiv preprint arXiv:2201.12085 (2022).

[66] David G. Lowe. 1999. Object recognition from local scale-invariant features. In Proceedings of the 7th IEEE International Conference on Computer Vision (ICCV’99). Vol. 99. 1150–1157.

[67] Muhammad Noman Malik, Huma Hayat Khan, and Fazli Subhan. 2017. Sustainable design of mobile icons: Investigating effect on mentally retarded user’s. Journal of Medical Imaging and Health Informatics 7, 6 (2017), 1419–1428.

[68] Mika V. Mäntylä, Kai Petersen, Timo O. A. Lehtinen, and Casper Lassenius. 2014. Time pressure: A controlled experiment of test case development and requirements review. In Proceedings of the 36th International Conference on Software Engineering, 83–94.

[69] Jan Meskens, Kris Luyten, and Karin Coninx. 2009. Plug-and-design: Embracing mobile devices as part of the design environment. In Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems. 149–154.

[70] Jan Meskens, Kris Luyten, and Karin Coninx. 2009. Shortening user interface design iterations through realtime visualisation of design actions on the target device. In Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’09). IEEE, Los Alamitos, CA, 132–135.

[71] Microsoft. 2021. Sketch2Code. Retrieved May 6, 2022 from https://sketch2code.azurewebsites.net/.

[72] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In Advances in Neural Information Processing Systems. 3111–3119.

[73] Mockup. 2021. Mockup.io—Present and Manage iOS Mockups. Retrieved May 6, 2022 from https://mockup.io/about/.

[74] Higinio Mora, Virgilio Gilart-Iglesias, Raquel Pérez-del Hoyo, and María Dolores Andújar-Montoya. 2017. A comprehensive system for monitoring urban accessibility in smart cities. Sensors 17, 8 (2017), 1834.

[75] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. In Proceedings of the 40th International Conference on Software Engineering. 165–175.

[76] Kevin Patrick Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. IEEE Transactions on Software Engineering 46, 2 (2018), 196–221.

[77] Fionn Murtagh and Pierre Legendre. 2014. Ward’s hierarchical agglomerative clustering method: Which algorithms implement Ward’s criterion? Journal of Classification 31, 3 (2014), 274–295.

[78] Marc Najork and Janet L. Wiener. 2001. Breadth-first crawling yields high-quality pages. In Proceedings of the 10th International Conference on World Wide Web. 114–118.

[79] Theresa Neil. 2014. Mobile Design Pattern Gallery: UI Patterns for Smartphone Apps. O’Reilly Media Inc.

[80] Javad Nejati and Aruna Balasubramanian. 2016. An in-depth study of mobile browser performance. In Proceedings of the 25th International Conference on World Wide Web. 1305–1315.

[81] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with REMAUI (T). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE’15). IEEE, Los Alamitos, CA, 248–259.

[82] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. 311–318.
Auto-Icon+

[83] Photoshop. 2021. Photoshop Apps—Desktop, Mobile, and Tablet. Retrieved May 6, 2022 from https://www.photoshop.com/en.

[84] Proto. 2021. Proto.io Prototyping Tool—Prototyping for All. Retrieved May 6, 2022 from https://proto.io/.

[85] J. Ross Quinlan. 1983. Learning efficient classification procedures and their application to chess end games. In Machine Learning. Springer. 463–482.

[86] Sanae Rosen, Bo Han, Shuai Hao, Z. Morley Mao, and Feng Qian. 2017. Push or request: An investigation of HTTP/2 server push for improving mobile performance. In Proceedings of the 26th International Conference on World Wide Web. 459–468.

[87] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2018. Examining image-based button labeling for accessibility in Android apps through large-scale analysis. In Proceedings of the 29th International ACM SIGACCESS Conference on Computers and Accessibility. 119–130.

[88] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 4510–4520.

[89] Thomas W. Sederberg and Rida T. Farouki. 1992. Approximation by interval Bézier curves. IEEE Computer Graphics and Applications 5 (1992), 87–88.

[90] Peter Selinger. 2003. Potrace: A polygon-based tracing algorithm. Potrace. Retrieved May 6, 2022 from http://potrace.sourceforge.net/potrace.

[91] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014).

[92] Sketch. 2021. Sketch Developer—File Format. Retrieved May 6, 2022 from https://developer.sketch.com/file-format/.

[93] Sketch. 2021. Sketch—The Digital Design Toolkit. Retrieved May 6, 2022 from https://www.sketch.com.

[94] Android Studio. 2021. Android Studio Home Page. Retrieved May 6, 2022 from https://developer.android.com/studio.

[95] Shamik Sural, Gang Qian, and Sakti Pramanik. 2002. Segmentation and histogram generation using the HSV color space for image retrieval. In Proceedings of the International Conference on Image Processing. Vol. 2. IEEE, Los Alamitos, CA, II.

[96] Kashyap Todi, Luis A. Leiva, Daniel Buschek, Pin Tian, and Antti Oulasvirta. 2021. Conversations with GUIs. In Proceedings of the 2021 Designing Interactive Systems Conference. 1447–1457.

[97] Fahui Wang. 2012. Measurement, optimization, and impact of health care accessibility: A methodological review. Annals of the Association of American Geographers 102, 5 (2012), 1104–1112.

[98] Xu Wang, Chunyang Chen, and Zhenchang Xing. 2019. Domain-specific machine translation with recurrent neural network for software localization. Empirical Software Engineering 24, 6 (2019), 3514–3545.

[99] Xiang-Yang Wang, Jun-Feng Wu, and Hong-Ying Yang. 2010. Robust image retrieval based on color histogram of local feature regions. Multimedia Tools and Applications 49, 2 (2010), 323–345.

[100] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. IEEE Transactions on Image Processing 13, 4 (2004), 600–612.

[101] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P. Bigham. 2021. Screen parsing: Towards reverse engineering of UI models from screenshots. In Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology. 470–483.

[102] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. IEEE Transactions on Software Engineering 42, 11 (2016), 1054–1076.

[103] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. ICONINTENT: Automatic identification of sensitive UI widgets based on icon classification for Android apps. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE’19). IEEE, Los Alamitos, CA, 257–268.

[104] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. UIED: A hybrid tool for GUI element detection. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 1655–1659.

[105] Shunguo Yan and P. G. Ramachandran. 2019. The current status of accessibility in mobile apps. ACM Transactions on Accessible Interactive Systems 12, 1 (2019), 1–31.

[106] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. UIS-hunter: Detecting UI design smells in Android apps. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion’21). IEEE, Los Alamitos, CA, 89–92.

[107] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, 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.
[108] 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.

[109] Dehai Zhao, Zhinchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: Vision-based workflow action recognition from programming screencasts. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE’19). IEEE, Los Alamitos, CA, 350–361.

[110] Dehai Zhao, Zhinchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seeomaly: Vision-based linting of GUI animation effects against design-don’t guidelines. In Proceedings of the 42nd International Conference on Software Engineering (ICSE’20). ACM, New York, NY.

[111] Hui Zhao, Min Chen, Meikang Qiu, Keke Gai, and Meiqin Liu. 2016. A novel pre-cache schema for high performance Android system. Future Generation Computer Systems 56 (2016), 766–772.

[112] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. 2021. GUIGAN: Learning to generate GUI designs using generative adversarial networks. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE’21). IEEE, Los Alamitos, CA, 748–760.

Received 30 July 2021; revised 11 March 2022; accepted 11 April 2022