Seenomaly: Vision-Based Linting of GUI Animation Effects Against Design-Don't Guidelines

Dehai Zhao
Zhenchang Xing*
dehai.zhao@anu.edu.au
zhenchang.xing@anu.edu.au
Research School of Computer Science
Australian National University

Chunyang Chen
chunyang.chen@monash.edu
Faculty of Information Technology
Monash University
Australia

Xiwei Xu
Liming Zhu†
xiwei.xu@data61.csiro.au
liming.zhu@data61.csiro.au
Data61, CSIRO
Australia

Guoqiang Li‡
li.g@sjtu.edu.cn
School of Software
Shanghai Jiao Tong University
Shanghai, China

Jinshui Wang§
jinshui.wang@anu.edu.au
Research School of Computer Science
Australian National University

ABSTRACT

GUI animations, such as card movement, menu slide in/out, snackbar display, provide appealing user experience and enhance the usability of mobile applications. These GUI animations should not violate the platform's UI design guidelines (referred to as design-don't guideline in this work) regarding component motion and interaction, content appearing and disappearing, and elevation and shadow changes. However, none of existing static code analysis, functional GUI testing and GUI image comparison techniques can "see" the GUI animations on the scree, and thus they cannot support the linting of GUI animations against design-don't guidelines. In this work, we formulate this GUI animation linting problem as a multi-class screencast classification task, but we do not have sufficient labeled GUI animations to train the classifier. Instead, we propose an unsupervised, computer-vision based adversarial autoencoder to solve this linting problem. Our autoencoder learns to group similar GUI animations by "seeing" lots of unlabeled real-application GUI animations and learning to generate them. As the first work of its kind, we build the datasets of synthetic and real-world GUI animations. Through experiments on these datasets, we systematically investigate the learning capability of our model and its effectiveness and practicality for linting GUI animations, and identify the challenges in this linting problem for future work.

*Also with Data61, CSIRO.
†Also with University of New South Wales.
‡Corresponding author
§Corresponding author

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380411

CCS CONCEPTS

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

KEYWORDS

GUI animation, Design guidelines, Lint, Unsupervised learning

ACM Reference Format:
Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: Vision-Based Linting of GUI Animation Effects Against Design-Don't Guidelines. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380411

1 INTRODUCTION

Graphical User Interface (GUI) is an ubiquitous feature of mobile applications. Mobile platforms provide GUI design guidelines to which the mobile applications run on the platforms are expected to adhere, for example, Android Material Design and iOS Design Themes. In addition to the guidelines about static GUI visual effects like color system and typography [5], many design guidelines are about GUI animations, such as sliding in/out menu or sheet, expanding/collapsing cards. Figure 1 shows two examples of GUI animation[1] guidelines in the Android Material Design: (a) regarding the use of a visible scrim with modal bottom sheets; (b) regarding how to reveal card information.

GUI animations, if done properly according to such design guidelines, make a GUI more appealing and more usable. But a design guideline documentation has so many rules, regarding many aspects of GUI design such as layout, color, typography, shape, motion, which make it nearly impossible for a developer to remember all these rules and properly adopt them in actual app development. For example, Figure 7 shows some real-application GUIs that violate some design-don't guidelines in Figure 1 and Table 1. Although

[1]We cannot show animated images in the paper. All GUI animation examples in this paper are available at https://github.com/DehaiZhao/Seenomaly.

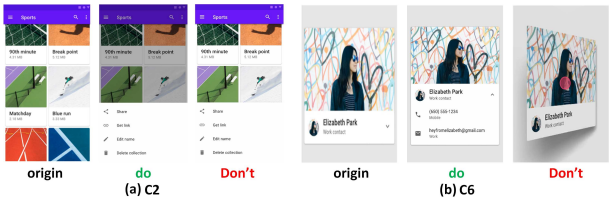

Figure 1: Examples of GUI Animation Dos versus Don'ts

such GUI design violations do not affect the functionalities of an application, they may result in poor user experience.

It is desirable to automatically validate the GUI animation of an application against design-don't guidelines to flag potential GUI design violations. This type of software tool is called lint or linter. Almost all major programming languages and platforms have corresponding lint tools, to name a few, the origin Unix lint for C, the famous FindBugs and CheckStyle for Java, and StyleLint for Cascading Style Sheet. These lint tools rely on human-engineered rules to detect bugs and stylistic errors.

Unfortunately, it is often not straightforward to convert GUI design-don't guidelines into programming rules, especially when they involve GUI animation effects. This is why many GUI animation guidelines have to be accompanied by illustrative do and don't examples. Furthermore, modern mobile platforms have very complex GUI style and theme systems (see for example Style and Theme of Android system). Developers can specify the desired style either declaratively in manifest file or programmatically in source code. They can also extend and customize existing styles and themes. All these complexities make it difficult, if not impossible, to precisely determine the actual GUI visual effects by static program analysis.

GUI testing [11, 13, 30, 36] can dynamically explores GUI behaviors of an application. However, GUI testing techniques, including the recently proposed deep learning based techniques [11, 36], focus on simulating the user interaction with the GUIs to trigger the app functionalities. Their effectiveness is evaluated with the code and GUI coverage. They cannot validate the visual effects of GUI designs. Moran et al. [24] recently proposes a computer-vision based method to detect the violations of an implemented GUI against the up-front UI design. Their technique examine only static GUI visual effects. However, GUI animations involve component movement, appearing and/or disappearing, and thus cannot be reliably detected by analyzing static GUI images (see examples in Figure 4 and the discussion in Section 4.1). Furthermore, their technique assumes that the two compared GUIs are largely similar but have minor differences which indicates the design violations. This assumption obviously does not hold for the actual GUIs (e.g., those in Figure 7) that violate certain design guidelines and the GUI animation examples illustrating the guidelines (e.g., those in Fig. 1).

In fact, the do and don't GUI animations illustrating a design guideline represent a class of conforming and violating GUIs. In this sense, we can regard the validation of a GUI animation against several design-don't guidelines as a multi-class classification problem: given a GUI animation, predict the design-don't guideline that this GUI animation likely conforms (or violates). Multi-class video classification [15, 28, 33] has been well studied in computer vision domain, and the recent advances of deep learning based methods have led to many breakthroughs. Unfortunately, we cannot adopt these multi-class video classification techniques to our GUI animation linting problem, because they are supervised learning methods which demand large amount of labeled video data. However, we have only a very small number of illustrative GUI animation examples for each design guideline. In machine learning terminology, our problem is a few-shot learning problem [31]. Manually collecting a large number of conforming and violating GUI animations for each design guideline would demand significant time and effort.

In this paper, we propose an unsupervised deep learning method to solve the GUI animation linting problem in the context of few shot learning. We first use automatic GUI exploration method [12] to build a large dataset of unlabeled GUI animations. Using this dataset, we train a vision-based GUI animation feature extractor to learn to extract important temporal-spatial features of GUI animations in an unsupervised way. Next, we use the trained feature extractor to map a few typical GUI animations that violate a design-don't guideline into a dense vector space. A GUI animation to be linted is also mapped into this vector space. We can then determine the design-don't guideline that this linted GUI animation most likely violates by a simple k-nearest neighbor (KNN) search of the most similar violating GUI animations. As KNN is a non-parametric method which does not require model training, our approach elegantly solves the challenge of few shot learning.

We implement our approach and use the tool to lint the GUI animations of Android mobile applications against nine Android Material design guidelines that cover diverse animation aspects, screen positions and UI components. We build a large-scale unlabeled real-application GUI animations from the Rico dataset [12] for training our adversarial autoencoder. Through experiments on a dataset of 9000 labeled synthetic GUI animations, we confirm the practical settings for applying our approach and the effectiveness of our model design over other model variants. Through experiments on a dataset of 225 labeled, distinctive real-world GUI animations, we show that our approach can accurately lint real-world GUI animations against a small number of GUI animation examples of design-don't guidelines. As the first work of its kind, our experiments also identify several challenging GUI animations (instantaneous GUI animations, simultaneous multiple component animations) which deserve further research.

We make the following contributions in this paper:

- To the best of our knowledge, our work is the first to lint GUI animation effects against design-don't guidelines.
- We propose the first deep-learning based computer vision technique for the GUI animation linting problem.
- The proposed model is trained by unsupervised learning and the unlabeled training data is easy to collect.
- Our experiments confirm the effectiveness of our approach on both synthetic and real-world GUI animations.

2 APPROACH

Our approach aims to lint the GUI animations of an application in response to the user actions (referred to as the linted GUI animation) against a list of the design-don't guidelines in a design-guideline documentation (e.g., Google Material Design). As illustrated in

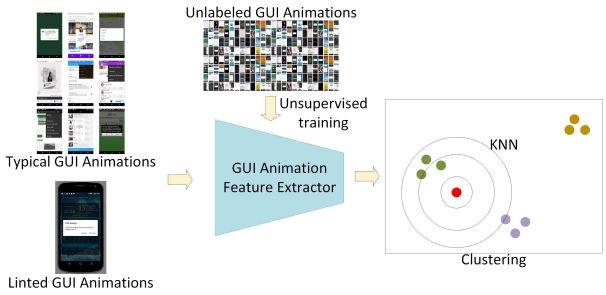

Figure 2: Approach Overview

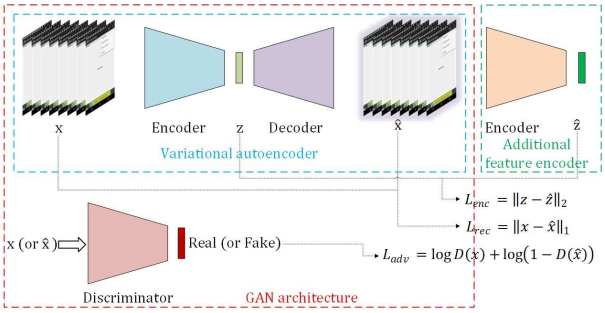

Figure 3: Model Architecture

Fig. 2, we formulate this linting task as a multi-class GUI-animation classification problem, and solve the problem by the k-Nearest Neighbor (KNN) search of the design-don't guideline that has the most similar GUI animation examples to the linted GUI animation. We design an adversarial autoencoder based feature extractor, which learns to extract abstract temporal and spatial feature representations of unlabeled real-application GUI animations in an unsupervised way, in the absence of a large number of the labeled GUI animations.

2.1 Input: Screencasts of GUI Animations

GUI animations can be triggered by users (e.g., slide in/out menu) or by applications (e.g.,. display snackbars). The GUI animation effect can be recorded in a GUI screencast, i.e., a sequence of GUI screenshots taken at a fixed time interval. A GUI screenshot is a 2-dimensional (height and width) image of the application's GUI.

2.1.1 Input Types. As shown in Figure 2, our approach has three types of input GUI animations. The first type is the linted GUI animation screencast that the developer wants to validate against a list of design-don't guidelines. The second type is a small number of the GUI animation screencasts that demonstrate the design-don't guideline (see examples in Figure 1), against which the linted GUI animation is compared. The second type of GUI animations is the labeled data for a design-don't guideline, and can be collected from the design-guideline documentation, online blogs discussing relevant design guidelines, and real applications. The third type is a large number of GUI animation screencasts that are automatically collected from real applications [8, 12]. These screencasts are unlabeled as we do not know whether and which design guideline they violate. We use this large amount of unlabeled data to train the unsupervised GUI animation feature extractor (see Section 2.2).

2.1.2 GUI Animation Capturing. We record a GUI animation screencast at 5 screenshot frames per second (5 fps). This recording rate can record significant screen changes during GUI animation, without recording many static frames in between the screen changes. The GUI animation screencasts are recorded in full RGB color. We record a GUI animation screencast for each individual user action, rather than a continuous screencast that may involve multiple user actions and also no-action periods. The recording starts after the user interacts with the application and the significant screen changes are detected, and it stops when the next user action comes or the screen remains static over 1 second. We set 5 seconds as the maximum recording duration, because we observe that an individual GUI animation is rarely longer than 5 seconds.

Based on Android material design, touch targets should be at least 48 x 48 dp (density-independent pixel). Therefore, we consider the screen as static if the two adjacent frames have pixel differences less than 48 x 48 dp (may translate into different numbers of physical pixels depending on the screen size and resolution. see Pixel Density). This threshold filters out minor GUI changes like the time changes or notification changes in the system bar. Note that we do not stop recording immediately after the first two static frames are detected, because some static frames may be recorded between the two screen changes during the GUI animation.

2.2 GUI Animation Feature Extractor

As the examples in Figure 1 and Figure 7 show, the GUI animations that violate the same design-don't guideline can be very different in the detailed GUI designs (e.g., contents and visual effects). This makes it impossible to directly measure the similarity of GUI animations with respect to a design-don't guideline in the high-dimensional screencast space. Instead, low-dimensional abstract temporal-spatial features of similar GUI animations with respect to a design-don't guideline must be extracted from the screencasts. Unfortunately, the amount of the examples that demonstrates the design-don't guideline (i.e., the labeled data) is not enough to effectively train a GUI animation feature extractor in a supervised way. However, the GUI animations that conform or violate certain design guidelines are present, albeit implicit, in the real applications We design an unsupervised GUI animation feature extractor which learns to extract the latent temporal-spatial feature representations from such unlabeled real-application GUI animations.

2.2.1 Model Architecture. Our feature extractor is based on deep generative models: variational autoencoder (VAE) and generative adversarial network (GAN). The core idea to infer the latent temporal-spatial feature representations by learning to generate GUI animation screencasts and discriminate the real GUI animation screencasts from the generated ones. As shown in Figure 3, our model consists of three components: a VAE that learns to reconstruct the input GUI animation screencast, a GAN that enhance the generation capability of the VAE by adversarial training, and an additional feature encoder network that learns more abstract feature representation from the reconstructed screencast. As the input screencasts are temporal-spatial data, the VAE encoder, the discriminator and

the feature encoder use 3D Convolutional Neural Network (3D-CNN) [33] to extract temporal-spatial features from the screencasts. A 3D convolutional kernel with $k * k$ spatial size and d temporal depth slides over a sequence of frames on three directions (height, width and depth). 3D-CNN has been successfully used for the action recognition tasks [29], which is similar to our linting task.

The VAE, GAN and feature encoder networks are jointly trained using unlabeled GUI animation screencasts, and the training is guided by three loss functions in different networks that aim to minimize the distance between the input screencasts and the generated screencasts, as well as between the latent feature vectors of these screencasts. These loss functions are the reconstruction loss L_{rec} for the VAE network, the adversarial loss L_{adv} for the GAN, and the encoder loss L_{enc} for the additional feature encoder. These three loss functions are combined together as the model loss $L = L_{rec} + L_{adv} + L_{enc}$ to optimize the networks jointly.

At the inference time, the VAE encoder and the additional feature encoder are used to obtain Z and $\hat{Z}$ for the input and the reconstructed GUI animation screencast, respectively, which are then concatenated to map the input screencast into the latent feature space. In this space, the similarity of the GUI animations can be determined by the L_2 distance of their feature vectors.

2.2.2 Variational Autoencoder Network. We adopt the VAE network because of its capability of dimensionality reduction [16], which meets our goal to extract low-dimensional temporal-spatial features from high-dimensional GUI animation screencasts. The VAE network consists of an encoder subnetwork (VAE_E) and a decoder subnetwork VAE_D. In our model, the input X is a GUI animation screencast as descried in Section 2.1. The encoder subnetwork is a 3D-CNN with batch normalization [33] and *ReLU* activation (ReLU(a)=max(0,a)). It encodes the input GUI animation screencast X into a low-dimensional latent vector Z. The decoder subnetwork VAE_D works in the same way as the generator in GAN [14], but we use 3D instead of 2D convolutional transpose layers [37]. It generates a GUI animation screencast $\hat{X}$ from the latent vector Z. $\hat{X}$ is referred to as the reconstructed screencast. We also use batch normalization and *ReLU* activation in VAE_D. Through this encode-reconstruct process, the VAE can be trained with unlabeled input data, which fits well with our data setting.

The learning objective of the VAE network is to minimize the difference between the input X and the reconstructed input $\hat{X}$. In this work, we compute L_1 distance between X and $\hat{X}$ as the loss function of the VAE network (referred to as reconstruction loss): $L_{rec} = \|X - \hat{X}\|_1 = \sum_{i=1}^{N} \|f_i - \hat{f}_i\|_1$, where f_i and $\hat{f}_i$ are the i-th screenshot in X and $\hat{X}$ respectively. By minimizing the reconstruction loss, the latent vector Z can capture the most important temporal-spatial features of the input X such that the reconstructed input $\hat{X}$ has the least information loss.

2.2.3 Adversarial Training by GAN. Although the VAE is generally effective in dimensionality reduction, the latent feature representation Z learned by the VAE alone is often not good enough to reconstruct the input screencast [19]. Inspired by adversarial autoencoder in [17, 20], we adopt the adversarial training to enhance the reconstruction of the input GUI animation screencast by the VAE. Adversarial training was introduced in GAN [14] which is a leading technique for unsupervised representation learning. A GAN consists of a pair of networks: a generator G tries to generate samples from a data distribution and a discriminator D tries to distinguish real samples and generated fake samples. The network is trained in a min-max game where the generator seeks to maximally fool the discriminator while simultaneously the discriminator seeks to determine the sample validity (i.e. real versus fake), and the two networks finally reach Nash equilibrium [25].

In our model, the VAE decoder plays the role of the generator G. The discriminator D is similar with the discriminator network in Deep Convolutional Generative Adversarial Network (DCGAN) [26], but our model uses 3D instead of 2D convolutional layers. This discriminator is a binary classifier to predict real or fake input. In our work, the real input is X and the fake input is $\hat{X}$. In the adversarial training setting, the generator is updated based on the classification output of the discriminator. The loss of the discriminator (referred to as adversarial loss) is the cross-entropy loss of the binary classification output: $Ladv = \log D(X) + \log(1 - D(\hat{X}))$.

2.2.4 Feature Encoder Network. In addition to the VAE encoder that learns the latent feature representation Z from the input screencast X, we add an additional feature encoder that learns an additional latent feature representation $\hat{Z}$ from the reconstructed screencast $\hat{X}$. The underlying intuition is that $\hat{Z}$ would be a more abstract feature representation than Z, because $\hat{X}$ is reconstructed from the most important features of X only. The feature encoder network E has the same network structure as the VAE encoder VAE_E, but the two networks do not share weights so that different feature representations can be extracted. For the training of E, we minimize the L_2 distance between Z and $\hat{Z}$ so that E can learn how to extract useful features from the reconstructed screencast. That is, the encoder loss function of E is $L_{enc} = \|Z - \hat{Z}\|_2$.

2.3 GUI Animation Linting by KNN Search

Given a GUI animation screencast, we want to determine if it violates some design-don't guidelines, and if so, which guideline. Our approach regards a design-don't guideline as a class of GUI design violation (see examples in Figure 7). As our GUI animation screencasts are recorded for each individual user action, they generally involve only one primary animation effect, and one design violation (if any)). Therefore, we solve the GUI Animation linting problem by a KNN search based multi-class classification.

Specifically, we use the VAE encoder and the additional feature encoder to map both the linted GUI animation and the GUI animation examples of design-don't guidelines into a low-dimensional latent feature space. Then, we find the top-k GUI animation examples that are the closet to the linted GUI animation in the feature space. The distance of the two GUI animations is computed by the L_2 distance their feature vectors v_1 and v_2, i.e., $\|v_1 - v_2\|_2$. We use the majority vote by the k-nearest GUI animation examples to determine the guideline that the linted GUI animation violates. If the majority vote has a tie, we compare the average distance of the GUI animation examples of the tie guidelines, and consider the guideline with the shortest average distance as the guideline that the linted GUI animation violates.

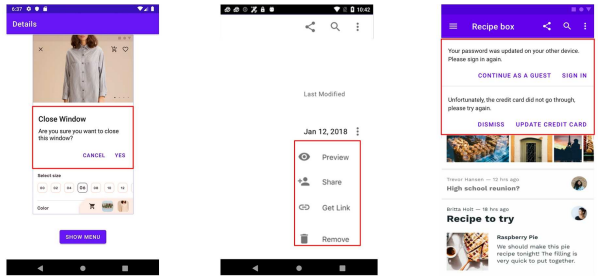

Figure 4: Design Violations Look Normal in Static Images

3 TOOL IMPLEMENTATION

Our current tool[2] takes GUI animation screencasts as the input. Each sample contains 8 frames and has the height and width 281 and 500 respectively. All input GUI animation screencasts are stored in the same format (animated GIFs). We use the setting of 8 screenshot frames because this setting is long enough to cover the duration of most GUI animations, and contains the least duplicate frames. The height and width setting are the same as that of the GUI animation screencasts in the Rico dataset [12]. In this way, we do not need to scale the size of the large amount of unlabeled GUI animation screencasts derived from the screencasts in the Rico dataset (see Section 4.2.3). For the screencasts that do not meet these requirements, our input processor normalizes them accordingly by sampling 8 frames (as many distinct ones as possible) and by scaling up/down the screencasts to 281×500.

4 EXPERIMENT DESIGN

We now describe the choice of design-don't guidelines used in our study, the datasets for training and evaluating our model, and the evaluation metrics used in our experiments.

4.1 The Design-Don't Guidelines

Table 1 summarizes the nine design-don't guidelines in Android Material Design we use to test the effectiveness and practicality of our approach. The nine guidelines cover common GUI animation aspects in Android Material design, including 5 elevation, 1 light and shadow, 4 content display, 3 component motion and 3 component interaction guidelines. They involve common Android UI components for user interaction, including dialog, snackbar, banner, app bar, sheet, menu, popup window, card, and these components may appear at the top, bottom, side, middle or anywhere on the screen. The guideline C4 may involve any UI components. The guideline C10 is not a specific design-don't guideline. Instead, it represents the normal GUI animations that do not violate any design-don't guidelines, to the best of our Android GUI design knowledge.

Furthermore, we select these nine guidelines because they all involve component motion, appearing and/or disappearing. As such, they cannot be reliably detected by analyzing static GUI images. Figure 4 shows three static GUI images after relevant GUI animations end: (a) a window pops up over a non-scrimmed background (C1); (b) a menu slides out without boundary shadow (C3); and (c) two banners are stacked at the top of the screen (C9). Without looking at the GUI animations that lead to these GUI images, these GUI images alone looks like normal GUI designs. But they actually violate the design-don't guidelines for GUI animations.

We carefully select the design-don't guidelines that have similar or related GUI animation effects to test the capability of our approach to distinguish them. For example, C1, C2 and C3 are all related to elevation. But C1 and C2 express elevation difference by scrimmed backgrounds, while C3 does so by shadows. C1, C2 and C3 are also different in terms of UI components involved and screen position affected: C1 involves dialogs which usually appear in the middle of the screen, C2 involves sheets appearing at the bottom of the screen, and C3 involves menu appearing at the side or popup window which may appear anywhere. C4 and C5 are also related. But C4 is generic about any material surface passing through another material surface, while C5 is specific about moving one card behind other card(s). Both C5 and C6 are card animation. But C5 is about card movement, while C6 is about card flipping. Both C2 and C7 involves something appearing from the bottom of the screen, but sheets in C2 is usually much larger than snackbars in C7. Both C7 and C8 are snackbar animation. But C7 is about snackbar blocking app bar, while C8 is about multiple snackbars.

However, we avoid to choosing repeating guidelines which may have little added value to the already-chosen guidelines in the experiments. For example, snackbars and banners are similar in terms of content display, but they appear in different places on the screen and also behave differently. As we already have two guidelines cover similar GUI animation effects of snackbars and banners, (i.e., C8 and C9 stack multiple snackbars or banners, but one changes the screen bottom and the other changes the screen top), we do not include the guideline "Place a banner in front of a top app bar" which essentially repeats another similar GUI animation effects of snackbars and banners, compared with C7.

4.2 Datasets

We build two labeled datasets (one with real-world GUI animations and the other with synthetic GUI animations) for evaluating our model, and a dataset of unlabeled real-application GUI animations for unsupervised training of GUI animation feature extractor.

4.2.1 Labeled Datasets of Real-World GUI Animations. We build a dataset of real-world GUI animations of the design-don't guidelines in Table 1 from three sources. First, we collect the illustrative examples of the selected design-don't guidelines in Android material design website (e.g., the don'ts in Figure 1). Second, we search the Web using the guideline descriptions to find more illustrative examples. For example, we find the online blog "AlexZH DEV" which contains some examples of the C1, C3 and C9 guideline in Table 1. Third, we collect the violations of the design-don't guidelines from the Android apps developed by the students in a Master-level Android development course in our school. Finally, we collect 25 real-world GUI animation screencasts for each design-don't guideline in Table 1. For the guideline C10, we randomly sample 25 normal GUI animations from the unlabeled dataset of real-application GUI animations described in Section 4.2.3. We use this *real-world* dataset to evaluate our model's performance in practice (RQ3). In order to

[2]The code is available on https://github.com/DehaiZhao/Seenomaly.

Table 1: Design-Don't Guidelines Used in Our Study

Design-don't Guideline	Animation Aspect	Screen Position	UI Component
C1: Lack of scrimmed background	Elevation	MIDDLE	Dialog
C2: Invisible scrim of modal bottom sheet	Elevation	BOTTOM	Sheet
C3: Lack of shadow	Elevation, Light and shadow	SIDE Anywhere	Menu PopupWindow
C4: Pass through other material	Elevation, Component Motion	Anywhere	Any component
C5: Move one card behind other card(s)	Elevation, Component Motion	Anywhere	Cards
C6: Flip card to reveal information	Content display, Component Motion	Anywhere	Card
C7: Snackbar blocks bottom app bar	Content display, Component interaction	BOTTOM	Snackbar, App bar
C8: Stack multiple snackbars	Content display, Component interaction	BOTTOM	Snackbars
C9: Stack multiple banners	Content display, Component interaction	TOP	Banners
C10: Normal GUI animation	Anything	Anywhere	Any component

test the capability boundary of our approach, we try our best to maximize the distinction between the selected GUI animations in terms of GUI content and visual effects (see examples in Figure 7), rather than having many similar ones to "inflate" the performance.

4.2.2 Labeled Datasets of Synthetic GUI Animations. We semi-automatically build a dataset of synthetic GUI design violations based on the collected real-world design violations and the real application GUI screenshots. First, we use the screen-change detection method described in Section 2.1.2 to detect the screen changes in each of the real-world GUI animation screencasts (including those of C10) in the real-world dataset, and then crop the changing GUI regions during the animation as a screencast. Then, we randomly select a real-application GUI screenshot from the Rico dataset [12], and replay the screencast of the cropped changing GUI regions on top of this screenshot. In this way, we obtain a synthetic GUI animation for the design-don't guideline of the original real-world GUI animation. When synthesizing GUI animations, we randomly apply a combination of scaling, brightness/contrast/saturation/hue jitter, lighting augmentation, color normalization techniques to augment the foreground GUI animations and/or the background GUI screenshots. The goal is to increase the diversity of the synthetic GUI animations and the difference between the synthetic GUI animations and the original real-world GUI animations.

The first author reviews the synthetic GUI animations and discard non-violation ones, such as those shown in Figure 5. In Figure 5 (a), a dialog is synthesized on top of a GUI screenshot that already shows a dialog, which is unrealistic. Furthermore, that GUI already shows the dialog over the dark background, so the synthetic GUI is not a violation of C1 (lack of scrimmed background). Similarly, the GUI animations in Figure 5 (b) and Figure 5 (c) do not violate C2 (invisible scrim of modal bottom sheet) and C3 (lack of shadow). In Figure 5 (d), a snackbar appears over a text window without a bottom app bar, so it is not a violation of C7 (snackbar block app bar). Note that the other 5 guidelines (C4/C5/C6/C8/C9) involve explicit component motion or interaction, which will always be a violation when that kind of component motion or interaction occurs. Finally, we obtain 1000 synthetic GUI animation screencasts for each design-don't guideline in Table 1. We use this *synthetic* dataset to investigate the impact of KNN settings (RQ1) and the ablation of model components (RQ2).

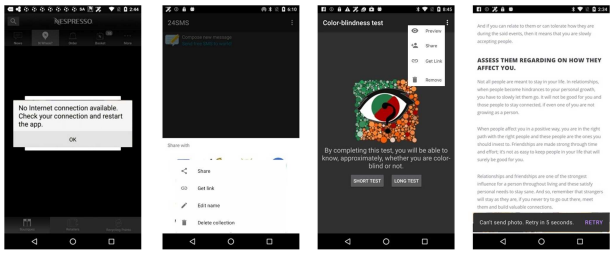

Figure 5: Non-Violation Synthetic GUI Animations

Note that due to the manual examination effort required, it is impossible to build a large enough synthetic dataset for supervised training of GUI animation feature extractor. Furthermore, as our RQ4 shows, the synthetic GUI animations have latent difference from the real-word ones, and thus the feature spaces of synthetic and real-world GUI animation do not match. This suggest that even we train a feature extractor using labeled synthetic GUI animations, it will not work effectively for encoding real-world GUI animations.

4.2.3 Unlabeled Dataset of Real-Application GUI Animations. We build a large-scale unlabeled dataset of real-application GUI animation screencasts from the GUI animation screencasts in the Rico dataset [12], which are collected from real human-app interactions and automatic GUI exploration. However, the duration of these screencasts spans from 0.5 second to 50 seconds. Many screencasts contain multiple user actions and no-action periods. We statistically analyze these screencasts to determine the appropriate maximum recording (5 seconds) duration and the stop-recording threshold (1 second) for our GUI animation capturing method describe in Section 2.1.2. We use this animation capturing method to playback each GUI animation screencast in the Rico dataset and re-record it into the GUI animation screencasts for individual user actions. Finally, we obtain GUI animation 91370 screencasts for training our GUI animation feature extractor.

4.3 Evaluation Metrics

We evaluate and compare the model performance for GUI animation linting by four metrics: Accuracy, Precision, Recall and F1-score.

The test data is a set of linted GUI animations that have no overlap with the GUI animations in the search space. For one GUI guideline class C, precision is proportion of GUI animations that are correctly predicted as C among all GUI animations predicted as C, recall is the proportion of GUI animations that are correctly predicted as C among all ground-truth GUI animations labeled as C, and F1-score is computed as $2\times precision_C \times recall_C/(precison_C + recall_C)$. Accuracy is an overall performance of the 10 guidelines, which is computed by the number of GUI animations correctly predicted by the model over all testing GUI animations.

5 EXPERIMENT RESULTS AND FINDINGS

We conduct extensive experiments to investigate the following four research questions:

- RQ1. How do different KNN settings (the number of GUI animation examples in the search space and the number of k-nearest neighbors searched) affect the classification performance?
- RQ2. How do different model components (i.e., VAE, GAN and additional feature encoder) in our GUI animation feature extractor affect the classification performance?
- RQ3. How well does our approach perform on linting real-world GUI animations against design-don't guidelines?
- RQ4. Are the latent feature space of synthetic and real-world GUI animations different? How does the difference, if exists, affect the cross-dataset design linting?

5.1 Impact of KNN Settings (RQ1)

5.1.1 Motivation. Our KNN search based GUI animation linting method has two important parameters that affect the classification accuracy and speed: 1) the number T of GUI animation examples of a design-don't guideline in the search space. 2) the number K of the nearest neighbors that vote for the design-don't guideline violated by the linited GUI animation. The RQ1 aims to study the impact of these two parameters on the classification performance.

5.1.2 Method. We use the synthetic dataset in this experiment because it allows the larger scale experiments than the real-world dataset. We experiment T={3,5,10,15,50,100,500} and K={1,3,5,10} (i.e., 28 T-K combinations). For each guideline C_i ($1 \le i \le 10$), we put T randomly selected GUI animations of C_i in the search space, and use the rest GUI animations as the linted GUI animations. We incrementally add more randomly sampled GUI animations to the search space, which simulates the practical situation in which more and more guideline-violating GUI animations are found and added to the search space over time. We use T=500 as a stress testing to investigate whether too many GUI animations in the search space will disturb the performance.

5.1.3 Results. Table 2[3] shows the accuracy and F1-score for the 28 T-K combinations. We can see that the T=3 and 10NN setting has the worst accuracy 0.51 and F1-score 0.16, much lower than all other settings. This is not surprising, because only 3 GUI animation examples for each guideline are in the search space, and considering the 10-nearest neighbors means that there are at least 7 examples whose design-don't guidelines are irrelevant to the linted GUI animation. These noisy samples highly likely mislead the classification decision, resulting in the performance drop. However, when considering less nearest neighbors (e.g., $K \le 5$) for T=3 or increasing the number of GUI animation examples ($T \ge 5$) in the search space, the accuracy becomes very close across different K settings for a specific T, with the slight (0.01-0.04) degradation when K increases. F1-score has the similar trend.

Table 2: Impact of KNN Settings on Synthetic Data

	Accuracy				F1-score			
T	1NN	3NN	5NN	10NN	1NN	3NN	5NN	10NN
3	0.64	0.63	0.63	0.51	0.53	0.45	0.42	0.16
5	0.67	0.66	0.65	0.63	0.57	0.49	0.45	0.39
10	0.70	0.70	0.70	0.67	0.59	0.56	0.54	0.48
15	0.71	0.71	0.71	0.69	0.60	0.58	0.57	0.52
50	0.77	0.77	0.76	0.75	0.69	0.67	0.66	0.63
100	0.79	0.79	0.79	0.78	0.73	0.71	0.70	0.67
500	0.90	0.88	0.87	0.86	0.83	0.80	0.79	0.77

Increasing T continuously improve the classification performance, from ~0.64 accuracy and ~0.5 F1-score for T=3 to ~0.79 accuracy and ~0.7 F1-score for T=100. Having 500 GUI animation examples per guideline in the search space does not disturb the classification decisions. Instead, it leads to a significantly performance boost to accuracy 0.86-0.90 and F1-score 0.77-0.83 for different K values. Although $T \ge 50$ leads to better performance, it may not be practical to collect 50 or more examples per guideline in practice. We see a trend of performance gap narrowing across K when T increases from 3 to 100 (accuracy difference between 1NN and 10NN down from 0.13 to 0.01, and F1-score difference between 1NN and 10NN down from 0.37 to 0.06). However, the performance gap becomes a bit wider again when T=500. This indicates that the 10-nearest neighbors still remain highly accurate, but become slightly more noisy when the search space has 500 examples per guideline, compared with less examples per guideline in the search space. However, when T increases from 3 to 500, the average search time increases from on average 0.53 second per linted GUI animation to 11.98 seconds. The average search time for T=10 or 15 is 1.32 and 1.88 seconds respectively.

> *The more GUI animation examples in the search space, the more comprehensive the search space becomes, and the better the classification performance is. When the search space have at least 5 examples, considering different numbers of nearest neighbors does not significantly affect the classification performance. The setting of 10 to 15 examples per guideline in the search space and searching for 3 to 5 nearest neighbors can achieve a good balance between the classification accuracy, the search time, and the effort to collect GUI animation examples.*

5.2 Model Component Ablation Study (RQ2)

5.2.1 Motivation. Our model (see Figure 3) consists of three components: a VAE network, a GAN architecture, and an additional

[3]Due to space limitation, we show the guideline-level performance results for all 28 T-K combinations in https://github.com/DehaiZhao/Seenomaly.

feature encoder. This RQ aims to investigate the impact and synergy of these components on the classification performance.

5.2.2 Method. We develop three variants of our model: the VAE-only, the VAE with the GAN architecture (VAE+GAN), the VAE plus the additional feature encoder (VAE+FE). These variants adopt the same model configuration as our tool. We train the three variants using the same unlabeled dataset for training our tool. The VAE-only and the VAE+GAN use only Z as the feature representation of GUI animations, but the VAE+FE uses both Z and $\hat{Z}$ as our method does. We run these three variants on the synthetic dataset in the same way as described in Section 5.1.2. That is, each variant has 28 experiments with different T-K combinations. Due to the space limitation, we report only the performance results for $T = 100$ and $K = 5$, including the overall performance and the guideline-level performance. Readers are referred to https://github.com/DehaiZhao/Seenomaly for the performance results of other T-K settings, which support the same findings as those shown in Table 3.

5.2.3 Result. Table 3 shows the precision, recall and F1-score for each guideline, as well as the average precision, recall, F1-score and the overall accuracy over all guidelines. The VAE network, as the core component of our unsupervised GUI animation feature extractor, already sets a very good performance baseline, with average precision 0.76, recall 0.64, F1-score 0.66, and overall accuracy 0.76. Just adding adversarial training through the GAN architecture or adding the additional feature encoder for extracting more abstract features do not affect the overall performance, but we see some variations in the guideline-level performance across VAE-only, VAE+GAN and VAE+FE. VAE+GAN and VAE+FE have a more balanced precision and recall than VAE-only. When combining the VAE with both the GAN architecture and the additional feature encoder (i.e., our model in Figure 3), we obtain better average F1-score and overall accuracy than VAE-only, with a more balanced average precision and recall than VAE-only. At guideline-level, our model achieve better F1-score (0.04-0.15) for 7 guidelines and worse F1-score (0.01-0.05) for 3 guidelines than VAE-only.

Looking into the guideline-level performance, our model performs the best for the guideline C4 (pass through other material) (F1-score 0.91) and C6 (flip card to reveal information) (F1-score 0.90). GUI animations related to these two guidelines involve salient component motion features, which are well learned by our model to classify these two types of GUI animations. However, our model does not perform so well for C5 (move one card behind other card(s)) (F1-score 0.68), which is also a component motion guideline. Moving one card among many others usually involve much more complex and diverse simultaneous multiple card animations, compared with the single component animation for C4 and C6.

Our model also perform very well for classifying normal GUI animations (C10) (F1-score 0.88). This indicates that although our model is trained in an unsupervised way, it can very well capture the data distribution differences between normal GUI animations and those with design violations. As a result, normal GUI animations are well separated from abnormal ones in the search space.

Our model performs much better for C2 (invisible scrim for modal bottom sheet) (F1-score 0.96) than C1 (lack of scrimmed background) (F1-score 0.68). Although both C2 and C4 are about

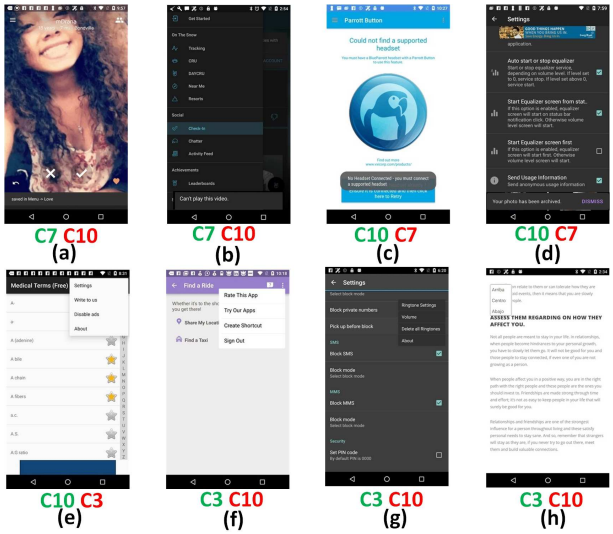

Figure 6: Incorrectly Classified GUI Animations (Green: Labeled Guideline, Red: Predicted Guideline)

scrimmed background, the modal bottom sheet in C2 appears gradually from the bottom of the screen, from which our model can learn much richer temporal-spatial features to recognize the animation of modal bottom sheet, than the instantaneous dialog popup in C1.

Our model performs reasonably well for C9 (stack multiple banners) (F1-score 0.70), but it performs not so well for another similar GUI animation (C8 stack multiple snackbars). For C8, the precision is still good (0.83), but the model misses many cases (recall only 0.42). Our analysis reveals that the appearing/disappearing of multiple snackbars or banners usually involves longer animation. Our animation capturing method (see Section 2.1) may drop some important animation frames of snackbar or banner appearing/disappearing. As a result, our model does not learn very well the complete features of the appearing/disappearing of multiple snackbars or banners. However, as banners are larger and contain more information and GUI components than snackbars, the model learns richer features to better classifying C9-related GUI animation than C8-related GUI animations. We will refine our GUI animation capturing method to handle long GUI animations more properly.

Our model perform poorly for another snackbar-related GUI animation (C7 snackbar blocks app bar) (F1-score only 0.31), as well as for C3 (lack of shadow) (F1-score only 0.32). Figure 6 shows typical examples that make the classification of these two guidelines very challenging. For C7, the classification becomes challenging when snackbars and app bar or background GUIs have very similar visual features, e.g., Figure 6 (a)/(b). In such cases, a C7-violating GUI animation may be classified as normal (C10). Another challenge in classifying C7 is that the model may erroneously recognize non-app-bar components as app bars. For example, a snackbar appears in front of a button in Figure 6 (c) and an image in Figure 6 (d), which are a normal animation (C10), but our model misclassifies them as C7. For C3, the key challenge lies in the fact that the shadow animation has only very minor screen changes. As such, the model may misclassify a normal menu with shadow as C3 (e.g.,

Table 3: Overall and Guideline-Level Performance of the Three Variants and Our Method for T=100 and K=5

	VAE-Only			VAE+GAN			VAE+FE			Our method		
Guideline	**Prec**	**Recall**	**F1**	**Prec**	**Recall**	**F1**	**Prec**	**Recall**	**F1**	**Prec**	**Recall**	**F1**
C1	0.55	0.52	0.53	0.82	0.98	0.90	0.62	0.57	0.59	0.67	0.69	0.68
C2	0.87	0.99	0.92	0.89	0.94	0.92	0.83	0.98	0.90	0.94	0.98	0.96
C3	0.83	0.13	0.23	0.75	0.31	0.44	0.45	0.29	0.35	0.56	0.23	0.32
C4	0.89	0.99	0.94	0.82	0.98	0.82	0.76	0.99	0.86	0.85	0.99	0.91
C5	0.80	0.67	0.73	0.76	0.72	0.74	0.62	0.66	0.64	0.67	0.69	0.68
C6	0.89	0.92	0.91	0.92	0.96	0.94	0.75	0.97	0.84	0.85	0.97	0.90
C7	0.27	0.41	0.33	0.30	0.19	0.23	0.33	0.25	0.29	0.29	0.34	0.31
C8	0.85	0.38	0.52	0.62	0.33	0.43	0.76	0.49	0.60	0.83	0.42	0.56
C9	0.85	0.52	0.65	0.63	0.43	0.51	0.79	0.53	0.64	0.75	0.65	0.70
C10	0.81	0.91	0.86	0.76	0.89	0.82	0.84	0.89	0.86	0.85	0.91	0.88
Average	**0.76**	**0.64**	**0.66**	**0.71**	**0.63**	**0.65**	**0.68**	**0.66**	**0.66**	**0.72**	**0.69**	**0.70**
Accuracy	**0.76**			**0.75**			**0.75**			**0.79**		

Table 4: Impact of KNN Setting on Real-World Data

	Accuracy				F1-score			
T	1NN	3NN	5NN	10NN	1NN	3NN	5NN	10NN
3	0.64	0.64	0.57	0.43	0.64	0.61	0.64	0.38
5	0.74	0.71	0.70	0.54	0.72	0.66	0.65	0.37
10	0.84	0.80	0.77	0.71	0.83	0.77	0.74	0.63
15	0.87	0.84	0.79	0.74	0.84	0.83	0.77	0.68

Figure 6 (e)) or misclassify a menu lack of shadow as C10 (e.g., Figure 6 (f)/(g)/(h)). These issues could be addressed by separating foreground animations from background GUI screenshots and then encoding them separately, rather than encoding them as a whole in our current model. We leave this as our future work.

> Our model can learn distinctive temporal-spatial features from real-application GUI animations in an unsupervised manner, based on which it can accurately lint GUI animations against a wide range of design-don't guideline involving very diverse animation effects, screen positions and GUI components. Our model currently has limitations on simultaneous multiple component animations and instantaneous component animations, but GUI animations with small screen changes or overlapping components with similar visual features is the most challenging GUI animations to classify.

5.3 Performance of Real GUI Linting (RQ3)

5.3.1 Motivation. Although synthetic data supports large-scale experiments in RQ1 and RQ2, it may have latent difference from real-world GUI animations, which may not be visually observable. Therefore, even though our model performs very well on synthetic data, we still need to test our model's capability of linting real-world GUI animations against design-don't guidelines.

5.3.2 Method. We use the same experiment method described in Section 5.1.2 for the experiments on the real-world dataset. We still experiment K={1,3,5,10}, but as the real-world dataset has

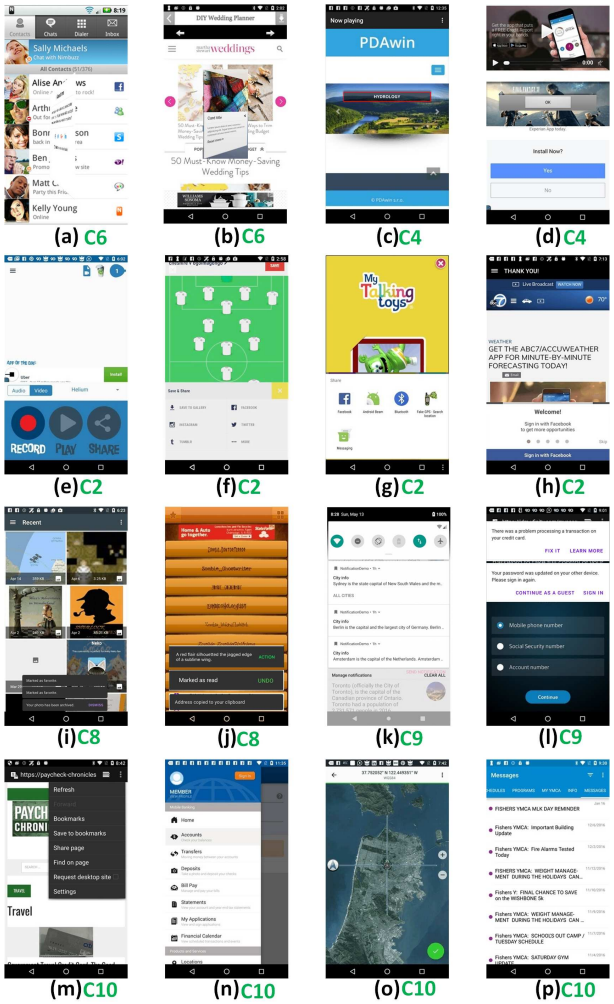

Figure 7: Correctly Classified GUI Animations

Table 5: Performance in the Real → Synthetic Context

	Accuracy				F1-score			
T	1NN	3NN	5NN	10NN	1NN	3NN	5NN	10NN
3	0.33	0.37	0.35	0.44	0.33	0.34	0.33	0.35
5	0.32	0.37	0.40	0.32	0.31	0.33	0.36	0.30
10	0.43	0.39	0.35	0.36	0.35	0.34	0.32	0.34
15	0.41	0.34	0.37	0.36	0.35	0.34	0.32	0.35

Table 6: Performance in the Synthetic → Real Context

	Accuracy				F1-score			
T	1NN	3NN	5NN	10NN	1NN	3NN	5NN	10NN
3	0.24	0.31	0.33	0.54	0.28	0.29	0.33	0.39
5	0.33	0.41	0.32	0.39	0.33	0.37	0.32	0.35
10	0.28	0.32	0.32	0.37	0.30	0.31	0.31	0.34
15	0.29	0.34	0.34	0.33	0.33	0.33	0.33	0.33

only 25 GUI animations for each guideline, we experiment only T={3,5,10,15}. So we have in total 16 T-K combinations in RQ3. We put at most 15 GUI animations per guideline in the search space, and this will leave us at least 10 GUI animations as the linted GUI animations. In addition to computing various evaluation metrics, we also manually examine the linting results, which is feasible as the real-world dataset is small, to understand the learning capability of our model qualitatively.

5.3.3 Result. Table 4 shows the accuracy and F1-score of for the 16 T-K combinations on the real-world dataset. We can make similar observations as those discussed in Section 5.1.3, in terms of the impact of different KNN settings on the model performance. The performance of our model on real-world data is generally better than that on synthetic data in the same T-K setting. This could be because the number of linted GUI animations is much smaller. However, as we try to maximize the distinction of the GUI animations (see examples in Figure 7) when building the real-world dataset, this performance results should not be because the linted GUI animations are too similar. Compared with the results on synthetic data, K value has larger impact on the performance for real-world data, especially when T is small and K is large.

Figure 7 shows some examples that our model classifies correctly. First, we can see that both the animated components and the background GUIs are very diverse in their content and visual/spatial features. In face of such data diversity, our model can still correctly classify them. Second, the linted GUI animations do not have to be similar to the GUI animation examples in the search space at the fine-grained design level (compare the examples (a)/(b) with the flipping-card example and compare the examples (e)-(h) with the model bottom sheet example in Figure 1) Third, our model has certain level of generalizability. For example, the example (n) shows a modal side sheet over a scrimmed background. Although it looks like a side menu without shadow (C3), our model correctly classifies it as normal. The examples (o) and (p) involve moving map and scrolling text, respectively. Our model also correctly classifies them as normal, which means that our model can place them at very different locations from those guideline-violating GUI animations.

> Our approach can accurately lint real-world GUI animations against a set of diverse design-don't guidelines, because it can learn abstract temporal-spatial features from GUI animations and ignore fine-grained GUI design differences.

5.4 Cross-Dataset Design Linting (RQ4)

5.4.1 Motivation. We observe the performance difference between linting the real-world GUI animations (Table 4) and linting the synthetic GUI animations (Table 2). We hypothesize that the latent feature space of real-world GUI animations and synthetic GUI animations has significant difference. We conduct cross-dataset design-linting experiments to validate this hypothesis.

5.4.2 Method. We use the same experiment method described in Section 5.3.2. That is, we experiment 16 T-K combinations: T={3,5,10, 15} and K={1,3,5,10}. In RQ4, we take GUI animations in one dataset as the examples in the search space and those in the other dataset as the linted GUI animations. We refer to this context as cross-dataset design linting, as opposed to within-dataset design linting in RQ1, RQ2 and RQ3. We have two cross-dataset contexts, denoted as real → synthetic and synthetic → real.

5.4.3 Results. Table 5 and Table 6 show the accuracy and F1-score for the 16 T-K combinations in the real → synthetic and synthetic → real contexts. First, we can see that the performance in both cross-dataset design linting contexts is significantly worse than that in within-dataset design linting (see Table 2 and Table 4). Second, we do not see the continuous performance improvement when T increases as in the two within-dataset contexts. The impact of increasing the number of GUI animation examples in the search space on the performance seems rather random. This indicates that the similar GUI animation examples are not consistently mapped to the close locations in the search space. Third, in the context of within-dataset linting, increasing K generally leads to performance degradation, which indicates that the most similar GUI animation examples appear at the top of k-nearest neighbors. So a larger K tends to include more irrelevant examples. In contrast, we observe the opposite results in cross-data contexts. In fact, the T=3 and K=10 setting in the two cross-dataset contexts has the best performance among all the T-K combinations, which is the exact opposite to the results in the within-dataset contexts. This indicates that the actually-relevant GUI animation examples are not the closest ones to the linted GUI animations in the search space. They can only be included when considering more k-nearest neighbors.

Although synthetic GUI animations are constructed using the cropped real-world GUI animations and real-application GUI screenshots, the combination of both in synthetic GUI animations may have latent "unrealistic" features which may not be visually observable by human inspector, for example, incompatible color systems, typography or shape. To our feature extractor which is trained from the real-application GUI animations, such latent unrealistic features would make synthetic GUI animations look like "anomalies" to the

real-world GUI animations. The reasonably good performance in the within-synthetic context (see Section 5.1.3) shows that our feature extractor can consistently map those anomalies in the search space, so that relevant anomalies are still close to one another. However, as anomalies to the feature extractor, synthetic GUI animations would be mapped into very different locations in the feature space from the relevant real-world GUI animations, which results in very poor cross-dataset linting performance.

> The feature space of synthetic GUI animations is very different from that of real-world GUI animations. Therefore, it is unrealistic to perform cross-dataset design linting. A feature extractor trained using synthetic GUI animations would not be effective in encoding real-world GUI animations.

6 RELATED WORK

Our work proposes a novel software linting problem. Lint, or a linter, is a static analysis tool that flags programming errors, bugs, stylistic errors, and suspicious constructs. The term originates from a Unix utility that examines C language source code. Nowadays, all major programming languages and platforms have corresponding lint tools. For example, Android Lint [1] reports over 260 different types of Android bugs, including correctness, performance, security, usability and accessibility. StyleLint [2] helps developers avoid errors and enforce conventions in styles. All such lint tools rely on human-engineered rules. Although such rules could be defined for some static GUI visual effects (e.g., too small font, too large gap, too dark background, lack of accessibility information), it is not straightforward to define rules for GUI animations involving complex component motion, interaction, appearing or disappearing. Furthermore, the complexity of style and theme systems in modern GUI frameworks makes it very difficult to precisely determine the actual GUI effects by static programming analysis.

Different from static linting, automatic GUI testing [4, 22, 30] dynamically explore GUIs of an application. Several surveys [18, 38] compare different tools for GUI testing for Android applications. Unlike traditional GUI testing which explores the GUIs by dynamic program analysis, these two techniques use computer vision techniques to detect GUI components on the screen to determine next actions. These GUI testing techniques focus on functional testing, and they are evaluated against code or GUI coverage. In contrast, our work lints GUI animations obtained at runtime against design-don't guidelines.

Recently, deep learning based techniques [11, 36] have been proposed for automatic GUI testing. In addition to GUI testing, deep learning has also been used for UI-design-to-code transformation [6, 23], phishing app generation [9, 10, 32], app localization [35] and app accessibility [7]. Zhao et al. [39] proposes a deep learning based method for extracting programming actions from screencasts. The goal of our work is completely different from these works. Furthermore, they all use supervised deep learning techniques, which requires large amount of labeled data for model training. In contrast, our work deals with a novel problem with only small-scale labeled data, and our work is the first to use unsupervised representation learning for GUI-related linting problem.

Our approach is inspired by deep learning based action recognition [15, 28, 33, 34]. In particular, we adopt the 3D-CNN [33] as our basic feature extractor. These methods performs well on natural scene videos (e.g. human action recognition), which have very different environment from our work (screencasts of GUI animations). In addition, these models are trained via supervised learning. As the first work of its kind, our work is fundamentally limited by labeled data, we motivate us to adopt unsupervised learning method to solve our linting problem. Our model is also inspired by anomaly detection with unsupervised learning [3, 17, 20, 21, 27]. As reviewed in [17], adversarial learning [20] is suitable for detecting anomaly data from videos, which is similar to our goal. The difference is that anomaly detection is a binary classification task (normal or abnormal), while our task is a multi-class classification, which is a much harder task for unsupervised representation learning. We connect an additional feature encoder with the GAN to satisfy the need of extracting abstract animation features while ignoring fine-grained GUI details for our specific GUI linting problem.

7 CONCLUSION

This paper investigates a novel software linting problem - lint GUI animations against design-don't guidelines. We innovatively solve the problem using multi-class GUI animation classification. In absence of sufficient labeled data for training the classifier, we design a novel adversarial autoencoder for unsupervised representation learning of GUI animations. From a large amount of unlabeled GUI animations that are automatically collected from real applications, our model learns to extract abstract temporal-spatial feature representations from GUI animations with different fine-grained GUI content and visual effects, based on which a simple k-nearest neighbor search can accurately "see" the anomaly of an unseen GUI animations against a small number of (5-15) GUI animation examples of design-don't guidelines for a wide range of animation effects, screen positions and GUI components. As the first work of its kind, we also contribute two labeled and one unlabeled datasets for future research, and unveil the challenges that deserve further research in this novel software linting problem.

8 ACKNOWLEDGMENT

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

REFERENCES

[1] [n.d.]. http://tools.android.com/tips/lint.
[2] [n.d.]. https://github.com/stylelint/stylelint.
[3] Samet Akcay, Amir Atapour-Abarghouei, and Toby P Breckon. 2018. Ganomaly: Semi-supervised anomaly detection via adversarial training. In *Asian Conference on Computer Vision*. Springer, 622–637.
[4] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 238–249.
[5] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery DC: Design Search and Knowledge Discovery through Auto-created GUI Component Gallery. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–22.
[6] 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*. ACM, 665–676.

[7] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *42nd International Conference on Software Engineering (ICSE '20)*. ACM, New York, NY, 13 pages. https://doi.org/10.1145/3377811.3380327

[8] 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 41st International Conference on Software Engineering*. IEEE Press, 596–607.

[9] 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* (2019).

[10] 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.

[11] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 296–306.

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

[13] Android Developers. 2012. Ui/application exerciser monkey.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

[15] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 1725–1732.

[16] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[17] B Kiran, Dilip Thomas, and Ranjith Parakkal. 2018. An overview of deep learning based methods for unsupervised and semi-supervised anomaly detection in videos. *Journal of Imaging* 4, 2 (2018), 36.

[18] Tomi Lämsä. 2017. Comparison of GUI testing tools for Android applications. (2017).

[19] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. 2015. Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300* (2015).

[20] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. 2015. Adversarial autoencoders. *arXiv preprint arXiv:1511.05644* (2015).

[21] Jefferson Ryan Medel and Andreas Savakis. 2016. Anomaly detection in video using predictive convolutional long short-term memory networks. *arXiv preprint arXiv:1612.00390* (2016).

[22] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 559–570.

[23] Kevin Moran, Carlos BernalCrdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *arXiv preprint arXiv:1802.02312* (2018).

[24] 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*. ACM, 165–175.

[25] Martin J Osborne and Ariel Rubinstein. 1994. *A course in game theory*. MIT press.

[26] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[27] Thomas Schlegl, Philipp Seeböck, Sebastian M Waldstein, Ursula Schmidt-Erfurth, and Georg Langs. 2017. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery. In *International Conference on Information Processing in Medical Imaging*. Springer, 146–157.

[28] Karen Simonyan and Andrew Zisserman. 2014. Two-stream convolutional networks for action recognition in videos. In *Advances in neural information processing systems*. 568–576.

[29] Khurram Soomro, Amir Roshan Zamir, and M Shah. 2012. A dataset of 101 human action classes from videos in the wild. *Center for Research in Computer Vision* (2012).

[30] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 245–256.

[31] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip HS Torr, and Timothy M Hospedales. 2018. Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1199–1208.

[32] Chongbin Tang, Sen Chen, Lingling Fan, Lihua Xu, Yang Liu, Zhushou Tang, and Liang Dou. 2019. A large-scale empirical study on industrial fake apps. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 183–192.

[33] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. 2015. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*. 4489–4497.

[34] Amin Ullah, Jamil Ahmad, Khan Muhammad, Muhammad Sajjad, and Sung Wook Baik. 2017. Action recognition in video sequences using deep bi-directional LSTM with CNN features. *IEEE Access* 6 (2017), 1155–1166.

[35] 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.

[36] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 307–317.

[37] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Robert Fergus. 2010. Deconvolutional networks.. In *Cvpr*, Vol. 10. 7.

[38] Samer Zein, Norsaremah Salleh, and John Grundy. 2016. A systematic mapping study of mobile application testing techniques. *Journal of Systems and Software* 117 (2016), 334–356.

[39] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: vision-based workflow action recognition from programming screencasts. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 350–361.