UI Screens Identification and Extraction from Mobile Programming Screencasts

Mohammad Alahmadi[1,2], Abdulkarim Khormi[1,3], Sonia Haiduc[1]
[1]Florida State University, Tallahassee, FL, United States
{alahmadi, khormi, shaiduc}@cs.fsu.edu
[2]University of Jeddah, Jeddah, Saudi Arabia
[3]Jazan University, Jizan, Saudi Arabia

ABSTRACT

Mobile applications demand is on the rise, leading to more programmers learning to develop or having to maintain this kind of programs. Developers often refer to online resources to find inspiration or answers to questions they have about mobile programming topics and screencasts are a popular resource. However, given the multitude of screencasts available, it can be difficult to quickly comprehend which of the many videos is relevant to one's needs.

We propose a novel approach, called *UIScreens*, which detects, extracts, and presents the most representative user interface (UI) screens embedded in mobile development screencasts. This could help developers quickly comprehend what an app displayed in a video is about, therefore saving time searching for useful videos.

UIScreens has been evaluated in two empirical studies on iOS and Android programming screencasts. The first study investigates the accuracy of our UI extraction and shows that our approach is able to detect and extract UI screens with an accuracy of 94%. The second is a user study with mobile app developers, who evaluated both the accuracy and the usefulness of UIScreens. They agreed that UIScreens is accurate and extracts representative UI screens from videos. They considered that the extracted UI screens are useful for understanding what a video is about and if it is relevant to a search. Our approach has been implemented as a free online tool.

CCS CONCEPTS

• **Software and its engineering** → *Documentation*; • **Computer vision** → *Image recognition*;

KEYWORDS

Program comprehension, Programming video tutorials, Mobile development, Software documentation, Deep learning, Video mining

ACM Reference Format:
Mohammad Alahmadi[1,2], Abdulkarim Khormi[1,3], Sonia Haiduc[1]. 2020. UI Screens Identification and Extraction from Mobile Programming Screencasts. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389265

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389265

1 INTRODUCTION

Smartphones are some of the most widely used devices today, with more than 2.5 billion users worldwide [56]. The two most popular app stores, the Apple App Store[1] and Google Play Store[2] host millions of smartphone applications that people use for a variety of tasks in their daily lives [5, 17, 27]. The increasing demand for these applications has spurred growth in mobile app development with more and more programmers learning to develop new mobile applications or having to maintain and evolve existing ones.

When learning about new concepts and technologies, debugging, or looking for answers to programming questions, online resources are developers' preferred documentation sources [9]. Screencasts are particularly on the rise [36, 37, 48], with tens of millions of videos available on many programming topics, hosted on platforms like YouTube[3]. Screencasts can be effective learning tools for mobile app developers due to their ability to offer in-depth, step-by-step explanations on a particular programming topic [36] and the fact that they show the mobile apps and their features in action. However, it can be often hard to find a screencast that addresses the topics, features, or app elements for which a developer might be searching for. This is due in part to the fact that the hosting platforms (*e.g.*, YouTube) display very limited information for each video, making it hard for developers to quickly determine if a video contains the app features, user interface (UI) elements, or mobile development topics that are relevant to their information need.

In this paper, we make a step towards addressing this problem by localizing, extracting, and presenting to the developer the most representative UI screens found in a mobile programming screencast. This can be seen as a *UI overview of a video*, which can help developers quickly comprehend what the apps developed in programming screencasts are about and if they are relevant to their information needs. We focus on the UI of an app, since it captures the essence of an application [8, 39, 42], by showcasing the features it provides in action.

Our approach for extracting the UI overview, called *UIScreens*, is based on a deep Convolutional Neural Network (CNN) which includes an image feature extractor and an object detector to locate UI screens within the frames of a programming screencast. The detected UI screens are then extracted and filtered such that only unique UI screens are kept. The approach was also integrated into a tool, which is freely available to use online[4].

[1]https://www.apple.com/ios/app-store/
[2]https://play.google.com/store
[3]https://www.youtube.com
[4]http://uiscreens.ddns.net/

We conducted an evaluation of UIScreens through two empirical studies. The first study focused on determining the accuracy of our approach in correctly locating UI screens in 1,000 iOS and Android programming videos from YouTube. The results indicated that UIScreens can precisely locate UI screens in screencast frames, achieving an accuracy of 94%. The second part of the evaluation involved a user study where 25 professional developers and computer science students were asked to assess the results of our approach based on quality and usefulness. The evaluation was done on a new set of 50 iOS and Android screencasts, not involved in the training of our approach. The results indicated that participants valued the extracted UI screens and found them appropriate and useful.

In summary, the main contributions of this paper are:

- The *first approach for locating and extracting UI screens from mobile programming screencasts*, providing a UI overview of a video. This can enable developers to quickly comprehend which are the main features of an app explained in a screencast and determine if the video is relevant to their information needs.
- An evaluation based on two empirical studies showing that the proposed approach is not only accurate, but also considered useful by developers.
- A freely available tool implementing our approach.
- A replication package[5] containing our complete dataset, results, and scripts.

The rest of the paper is organized as follows: Section 2 introduces the main architectures and approaches for image analysis used in our work, Section 3 introduces our approach and its components, Section 4 describes the two empirical studies we performed, Section 5 discusses the threats to the validity of our results, Section 6 presents an overview of the related work, and finally Section 8 concludes the paper.

2 BACKGROUND

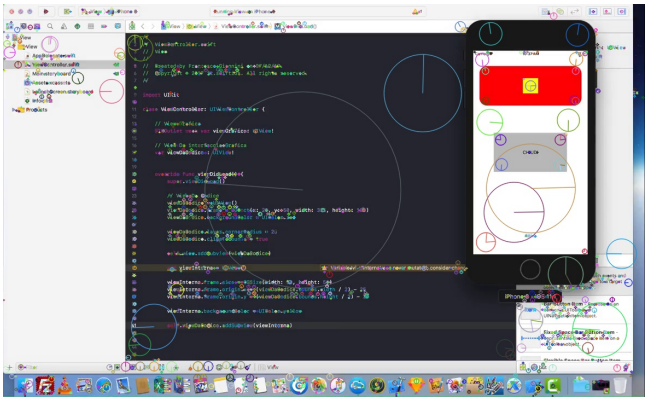


Figure 1: Keypoints extraction in a video frame using SURF

Using algorithms and techniques from other fields has proven extremely beneficial to software engineering research over the past couple of decades, with countless applications addressing various software engineering problems [12, 35, 50, 55, 59]. Deep learning algorithms have been particularly successful and their application in software engineering research has soared over the past few years [20, 31, 61, 62], including in work analyzing software development screencasts [43, 44] and GUI prototyping [38]. Given their proven benefits and the nature of our problem, we make use of the deep architecture of Convolutional Neural Networks (CNN), first introduced and most widely used in the field of computer vision [51]. We want to underline the fact that, while the backbone architectures we use are not novel, their training and application for generating UI-focused overviews of programming screencasts is.

The rest of this section introduces the specific techniques we use and their backbone architectures, which are at the core of our approach.

2.1 Image Feature Extraction

In the field of computer vision, describing an image by detecting its main features or so-called "keypoints" using *image feature extraction* techniques has been successfully applied to solve several image recognition challenges [6, 33]. In our approach, we use an image feature extractor in order to determine distinctive features in the frames of a mobile programming screencast and then compare consecutive frames based on these features in order to determine if they contain distinct information or not. If two consecutive frames are very similar, only one of them will be kept (more details about how we implement this in our approach will be presented in Section 3). This step is essential for reducing the number of frames analyzed to just those that contain unique information, as previous work has shown that programming screencasts contain many more duplicate frames than other types of videos [13].

As a general rule for feature extraction in images, the extracted features have to be distinctive and robust (*i.e.*, invariant to rotation, transformation, and scaling). In UIScreens, we make use of SURF for feature extraction. Speeded Up Robust Features (SURF) [6] is one of the best approaches towards extracting distinctive and robust features for images. Figure 1 shows an example of the features detected by SURF in a frame from a mobile programming screencast.

In order to determine the similarity between two frames and identify duplicates using SURF, three steps are performed. First, a Fast-Hessian detector is used to find the keypoints in distinctive areas (*e.g.*, blobs and corners). Then, a robust-to-noise feature vector (aka descriptor) is assigned for each keypoint identified in a frame. This descriptor has to be distinctive for the frame. To extract the descriptor, SURF assigns an orientation that is invariant to rotation for each keypoint by computing Haar wavelet response. Lastly, the similarity between two frames is determined by matching their descriptor vectors. If this similarity is above a threshold, one of the images is considered redundant and is discarded.

2.2 Object Detection based on Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have been widely and successfully used in solving several computer vision challenges such as image recognition, classification, and detection [29, 54, 67]. Similar to other types of artificial neural networks, CNNs for these types of tasks typically consist of multiple layers, including: (i) an

[5]https://zenodo.org/record/3743842

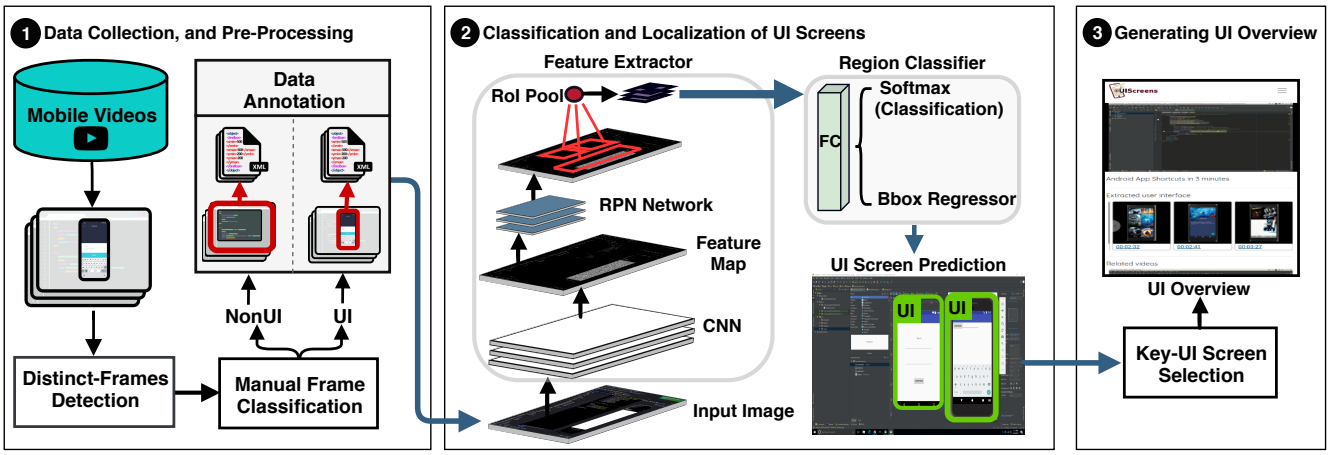


Figure 2: An overview of our approach and its main components

input layer that is typically an image with a three denominational pixel matrix (*i.e.,* RGB), (ii) convolution layers, where most of the mathematical computations occur to extract features from the input image through the use of filters, and (iii) fully connected layers, which are used to classify the input image based on the extracted low dimensional feature maps, typically by utilizing a softmax activation function. This function outputs the probability that an input image belongs to one of the pre-defined classes. After each convolution layer, there are typically a non-linear unit and a pooling layer. One of the most commonly used non-linear activation functions is Rectified Linear Units (ReLU) which can simply be defined as $f(x) = max(0, x)$ (*i.e.,* it outputs 0 for a negative input). The max pooling layer is used to downsample the feature map while keeping the salient features.

One of the successful applications of CNNs in computer vision has been in the area of object detection, where recent algorithms have shown impressive performance in finding objects inside an image or a scene. Most state-of-the-art object detection methods utilize CNNs to not only classify an object into a specific category but also to find a precise location of that object inside the image [24, 32]. Given their prior record on the object detection and localization task, we make use of CNNs to identify and precisely locate UI screens inside the frames of mobile programming screencasts.

There are two main approaches for detecting an object inside an image using CNNs: region-based and region-free. In our work, we utilize a region-based approach, namely Faster R-CNN, which is faster than other region-based approaches [51] and more accurate than region-free approaches [25, 63].

Region-based CNN object detection: Region-based approaches for object detection create a number of potential regions of interest in an image where an object is more likely to be located [60]. This selective search approach reduces the computational cost compared to processing all parts of an image. The features of the potential regions are extracted using a CNN and eventually classified by the output layer of the network using a softmax unit. Several region-based approaches have been proposed over the last few years. The earliest approach, called Region-based CNN (*R-CNN*) detects an object inside an image by running a CNN classifier on each proposed region [19]. Forward propagation is performed to extract the low dimensional feature map from each region. Each region is then classified using a Support Vector Machine. The bounding box of the potential location of the object is then adjusted to the position of the object using a linear regressor. Because R-CNN runs the CNN on every region, it is very slow in the object detection task. *Fast R-CNN* is a follow-up approach that was intended to optimize the performance of R-CNN [18]. Fast R-CNN reduces the computation cost by running the CNN on the entire image just once, rather than feeding each region independently to the CNN. The Region of Interest is identified after that, on the feature map of the entire image that was generated using the CNN. The Region of Interest (RoI) pooling layer is used to adjust the size of the region to a fixed size, leading to a representation of the region using feature vectors of that size. These fixed-size boxes are classified using a fully connected layer. While a softmax layer is used for the classification, the offset values adjust the bounding box. *Faster R-CNN* is the third and the most influential iteration of region-based CNN approaches [51]. Faster R-CNN optimizes the performance even further by using a Region Proposal Network (RPN) as an alternative to the selective search algorithm. The RPN creates region proposals that are adjusted using an RoI pooling layer and eventually classified with a tighter bounding box. Faster R-CNN outperformed all previous R-CNN iterations and has inspired several works in the field of object detection [53, 66, 68]. We use Faster R-CNN as the object detector in our approach.

3 APPROACH

In this section we describe UIScreens, our approach for extracting UI overviews of mobile programming screencasts. Figure 2 offers a general overview of our approach, which consists of three major steps. During the first step, video frames are collected from a set of mobile programming screencasts, and frames containing redundant information are removed, in order to speed up the processing time and ensure the diversity in our dataset. After that, the remaining frames are manually labeled as either containing UI screens or not. The frames containing UI screens are then manually annotated

with a bounding box that frames the UI. This data is then used for training a deep network based on the Faster R-CNN architecture which is then able to predict the presence and location of UI screens in unseen mobile screencasts. The last steps reduces the predicted set of UI screens to a smaller set of key UI screens, which consists the final UI-focused overview that is displayed to the user. The following subsections describe each of these steps in more detail.

3.1 Data Collection and Pre-Processing

3.1.1 Collecting Mobile Development Screencasts. Our approach, UIScreens, requires training data in order to build a prediction model that is able to detect and localize UI screens in mobile programming screencasts. This data, however, only needs to be collected once, after which the constructed model can be used on any number of new, unseen video tutorials.

In order to provide training (and testing) data for our approach, we manually collected a set of 1,000 mobile programming screencasts from YouTube[6]: 500 iOS tutorials and 500 Android tutorials. We aimed to collect a diverse set of videos in order to train a robust prediction model, able to recognize UI screens in a variety of circumstances. To ensure diversity in our dataset, we collected videos that: (i) display a variety of UI emulators for different mobile devices (*e.g.*, iPhone 6, iPhone X, iPad, Nexus 4, Google Pixel, etc.) and different screen sizes, (ii) display different IDEs and different programming languages specific for mobile development, such as Swift and Objective C for iOS, Java for Android, as well as cross-platform programming languages such as C# and Kotlin[7], (iii) display different phone orientations (portrait and landscape) in emulators, and (iv) cover a wide variety of mobile application types, such as puzzle, animation, quiz, augmented reality, etc. In addition, we also ensured that we collected no more than five videos per channel. On average, the length of the iOS and Android videos are $\sim$ 13 and $\sim$ 16 minutes, respectively.

It took a total of 83 hours to collect these videos manually following the above criteria. The videos were collected and validated by two authors who have a strong background in mobile development. We used the youtube-dl[8] API tool to download the videos to our server.

3.1.2 Distinct-Frames Detection. Previous work has found that programming tutorials are generally much more static than other types of videos [13], meaning that it is common for the content displayed on the screen to remain the same for longer periods of time during the screencast. This is due to the fact that programming tutors often spend some time explaining a piece of code or a programming concept displayed on the screen, in which time there is no noticeable change in the video frames. Therefore, many frames in a programming screencast are duplicates of each other. In consequence, the next step in our approach deals with eliminating duplicate frames and keeping only distinct ones for the analysis. This serves several purposes: first, it allows for a greater diversity in our dataset, ensuring that the deep learning object detector used in our next step has a more diverse set of frames to learn from and to be tested on; second, it reduces the unnecessary computation time that would be spent when training a model on similar images, having the same features; third, given that manual labeling of frames is needed for the next steps in our approach, this also reduces the amount of human effort involved.

Table 1: Statistics of our dataset before and after applying our approach to remove duplicate frames (in number of frames, at 1fps extraction rate)

	Mobile OS	Min.	Mean	1st Qu.	Median	3rd Qu.	Max.
Before	iOS	42	773	394	675	1032	2345
	Android	39	962	364	576	891	8259
After	iOS	1	64	24	50	86	366
	Android	1	57	25	42	71	455

We start by extracting one frame per second from each screencast, as it is custom when analyzing programming video tutorials [47, 65] and use the FFMPEG[9] off-the-shelf tool for this task. Let us then denote a video as V and a frame as f where V = $\{f_1, f_2, \ldots, f_n\}$, and n is the number of seconds in that video. We apply the feature descriptor SURF (see Section 2.1) on all n frames in the video to extract keypoints in each frame and the feature vectors that describe them. Then, given two neighboring frames f_1 and f_2, for each keypoint $k_{f_1,i}$ in frame f_1, we find the best matching keypoint $k_{f_2,j}$ and the second-best matching keypoint $k_{f_2,l}$ in f_2 based on the Euclidean distance between their features. If the Euclidean distance between $k_{f_1,i}$ and $k_{f_2,j}$ is smaller than 75% of the distance between $k_{f_1,i}$ and the $k_{f_2,l}$ (*i.e.*, the best matching point is significantly closer than the second-best match) then the pair of keypoints ($k_{f_1,i}$,$k_{f_2,j}$) is considered a strong match and added to the set of matching keypoints $m_{1,2}$ pairs for f_1 and f_2. The threshold of 75% for the distance was determined empirically, based on testing different values between 50% and 95%. This process is repeated for all keypoints in f_1, and at the end, the number of matched keypoints in $m_{1,2}$ represents the similarity measure between frames f_1 and f_2 (*i.e.*, the more similar keypoints in $m_{1,2}$, the more similar the two frames are). In other words, we obtain the similarity percentage between two frames by dividing the number of matched keypoints by the maximum between the total number of keypoints in each of the two frames. Then, if the similarity is greater than a given threshold, we consider that the frames contain mostly duplicate information and one of them can be discarded without losing relevant information.

Choosing the similarity threshold can impact the results of this step. Choosing a high threshold can result in many very similar frames being kept in the dataset, which can lead to overfitting when training our network and also increases the human effort when manually labeling the dataset. Choosing a low threshold can lead to removing frames that contain unique and relevant information that could be useful for training the model. Therefore, finding the right balance is important. We performed experiments involving various thresholds between 50% and 95% and found that a similarity threshold of 80% between frames offered a good compromise (together with the distance between keypoints of 75%, as mentioned above). When two frames are found to be similar based on these criteria, we remove the first frame and keep the last one, as it contains the latest version of the information. For example, assume a video remains

[6]https://www.youtube.org/
[7]https://kotlinlang.org
[8]https://github.com/rg3/youtube-dl
[9]https://www.ffmpeg.org/

static for its first 60 seconds while the tutor explains a piece of code. In this case, we remove f_1 through f_{59} and only keep f_{60}.

Table 1 shows detailed statistics about the collected number of frames (at 1 frame per second) before and after applying our approach for removing duplicate frames. The duplicate frame removal step resulted in a drop in the total number of frames from ~ 732k to only ~ 61k frames. The average sizes of the remaining sets of frames for iOS and Android screencasts are ~ 60 frames. All the remaining ~ 61k frames were then manually classified by two of the authors as either containing UI screens or not. The UI-containing frames were then further manually annotated by the two authors with bounding boxes that surround the UI screens appearing in them. These annotated frames represent the training data for our approach. The manual classification and annotation step is described below.

3.1.3 Manual Frame Classification and Annotation. In order to extract UI overviews of mobile programming screencasts, our approach first needs to be able to distinguish the frames that contain UI screens from those that do not. For this purpose, we need labeled training data exemplifying both frames containing UI screens and frames that do not contain such elements.

In order to obtain this data, two authors manually classified the remaining 60,865 distinct frames after duplicate frame removal into one of two categories:

- **User Interface (UI):** This category includes all frames that contain at least one fully visible mobile UI screen. The UI screens could be appearing in a layout editor page, interface builder, emulator, etc.
- **Non-User Interface (NonUI):** This category includes all frames that do not contain any fully visible UI (*i.e.*, they contain no mobile UI screen or mobile UI screens that are obstructed by other windows).

The two authors classified and verified these frames as follows. First, a supporting tool was built to make the classification faster. The tool displayed one frame at a time through a GUI that had four main buttons which allowed the authors to classify the frame as UI or NonUI and to move to the next or previous image. All frames were classified by one author and validated by another author. For easing navigation to a specific image during the validation process, a search bar was integrated into the tool. The two authors disagreed on 20 frames out of the 60,865 frames, and reached an agreement for each of them after a discussion. The disagreements arose when a UI was just slightly obstructed by another window. In those cases, one author considered the frame as UI, while the other considered it as NonUI. In the end, the agreement was to consider all these cases as NonUI, since we are interested in detecting non-obstructed, fully-visible UI screens for the final UI overview.

The total number of frames that were classified as UI by the authors was 5,695 for Android and 9,025 for iOS out of the total of 28,698 and 32,167 frames, respectively. On average, 14 frames were classified as UI from each video.

Our end goal is to extract the UI screens from the frames that contain them in a mobile programming screencast and then display the most representative ones as a UI overview of the video. To achieve this, our approach needs to also be able to locate where exactly the UI screens are found in the video frames and then extract only those portions of each image. This requires manually annotating the video frames in the UI category with the precise position of the UI screens they contain. More precisely, we annotate each UI screen with the coordinates of the bounding box that encloses it: $(x_{min}, y_{min}, x_{max}, y_{max})$, where x_{min} and y_{min} denote the upper left corner of the box and x_{max}, y_{max} denote its lower right corner. The NonUI frames are also annotated, since our model needs to also learn the features of a frame without a UI screen. However, we perform this annotation automatically and set the bounding box of NonUI frames to the entire frame (*i.e.*, (x_{min}, y_{min}) = (0,0) and (x_{max}, y_{max}) = (width, height)). Using this kind of annotated data, our model will be able to output the class name (either UI or NonUI) and the bounding box information for each input frame. Then, we use this information to isolate the UI frames and then extract the UI screens they contain based on the bounding box coordinates predicted by our approach.

Given the significant effort associated with manual labeling, we select a subset of the UI and NonUI frames for the bounding box annotation. Randomly selecting these frames from the entire dataset might result in unbalanced samples with respect to the number of videos in our dataset. That is, with random selection, we could obtain more UI frames from one video and none or few UI frames from other videos. To ensure diversity and uniformity, we randomly select one frame at a time from each collected video as follows. Given a set of videos V = $\{v_1, v_2, \ldots, v_n\}$, we select a random UI_i frame and a random $nonUI_i$ frame from each video v_i and add them to our selected set. We apply and repeat this process until we have a total of 2,000 UI and 2,000 NonUI frames selected, distributed fairly across the videos. Our goal is to have a diverse frame set from all the videos and selecting small samples from each video. On average, we select two random UI frames and two random NonUI frames from each video. Given that we have 500 iOS and 500 Android screencasts, the result of this process is a total of 1,000 iOS UI frames, 1,000 Android UI frames, as well as 1,000 NonUI iOS frames and 1,000 NonUI Android frames. Note that this way we also handled the class imbalance problem, preventing issues such as the model being biased toward the dominant class while working poorly for the minority class [21].

The manual annotation of each UI screen with its location was performed by two of the authors, using the cloud-based Dataturks[10] tool. All UI frames were annotated by one author and then the annotations were verified by another author.

While the manual annotation of UI frames is a tedious and time-consuming task, *it only needs to be performed once*, when gathering training data. The resulting trained model can then be applied over and over again on new videos, without the need for additional manual labeling. We also note that manually labeling data is often a necessary step when using supervised learning approaches.

The annotated UI and NonUI frames represent the input for the next step in the approach, which trains the object detector.

3.2 Classification and Localization of UI Screens

Our goal is to accurately identify and locate UI screens embedded in mobile development screencasts, such that they can be automatically extracted to produce UI overviews of the videos. To accomplish

[10] https://dataturks.com/

this, we aim to train a model with the distinguishing features of UI and NonUI frames and the location of UI screens within them. To detect and extract the features from the images, there are two main approaches. The first, intuitive approach would be to find hand-crafted features that describe the UI frames and train a classical machine learning algorithm such as Support Vector Machines, Naive Bayes, etc. to learn based on these features. However, this is impractical considering the large amount of features that could be considered for an image. Instead, the state-of-the-art approach is to automatically extract features using a deep Convolutional Neural Network (CNN) that considers a large number of parameters distributed across several layers. This approach has been used with impressive results in various computer vision tasks [22, 26, 54], and has also been successfully employed in software engineering tasks [38, 43, 44, 69]. To this end, we adopted a CNN-based object detector to extract the features of the UI screens.

In particular, we fine-tune the architecture of *Faster R-CNN* [51] described in Section 2.2 to locate UI screens based on features extracted using the backbone feature extractor *Inception-Resnet V2* [57]. Inception-Resnet V2 automatically extracts relevant spatial features of the UI regions that map to their coordinates (*i.e.*, their bounding boxes). We used Inception-Resnet V2 as a backbone network [57], since it was shown to outperform previous Inception and Inception-Resnet architectures [22, 26, 58] and also outperformed many other architectures in the ImageNet Large-Scale Visual Recognition Competition (ILSVRC) [52]. In addition, Inception-Resnet V2 has also been proven efficient in analyzing programming video tutorials [69]. We train the Faster R-CNN with Inception-Resnet V2 as it has been shown that this combination outperformed several other object detectors with various backbone networks in an object detection task [25]. Although Faster R-CNN was designed to work with natural scene images, we believe it can work even better for recognizing mobile UI screens in programming video tutorials because they have more predictable features and outlines/shapes than natural scenes do (*e.g.*, the dark color and unique shape of the phone emulator, a camera located at the top of the phone emulator, phone buttons at the bottom of the emulated phone, etc.).

During the training of our deep network, we follow the process depicted in the second phase in Figure 2, as follows: 1) A feature map is obtained for each input frame using deep convolutional layers that use a large number of parameters (*i.e.*, there are 164 convolutional layers in the Inception-Resnet architecture). 2) The feature map is then fed into a Region Proposal Network (RPN), which in turn generates regions of interest (ROIs) that have a high chance of containing objects. RPN considers in total nine different scales and aspect ratios, which is important because objects could have different shapes. 3) Finally, the extracted feature map is cropped based on the size of the RoIs obtained using the second step. The cropped batches must have the same size before feeding them into the detection network. Thus, RoI pooling is applied to obtain fixed-size batches. Then, the CNN-based region classifier takes the pooled feature of each region of interest as an input batch and uses Fully Connected (FC) layers to output the object's class and its bounding box coordinates.

Implementing our Object Detector: Our CNN-based object detector was trained end-to-end using back-propagation where the weights get adjusted using Stochastic Gradient Descent (SGD) with momentum optimizer [49]. We avoided overfitting by using the learning rate schedules technique, which adjusted the learning rate during the training process. The initial learning rate we used during our training was 3×10^{-4}, to guarantee the convergence of the training phase. Additionally, we used data augmentation techniques to make our model more robust and generalizable. An example of our data augmentation techniques involved randomly flipping our images horizontally during the training phase. The corresponding bounding boxes were also updated accordingly (*i.e.*, flipped horizontally). We also randomly scaled some of our training regions to make sure that the model learns the features of UI screens with different sizes.

The *batch size* is the total number of training samples that will be passed through the network at once. The batch size can be adjusted based on the computer hardware performance such as the GPU memory, as well as the input image size. We set the batch size to 16 based on the limitations of our hardware configuration. We then resized the input image dimensions to the default values of 600 X 1024 pixels, using a stride of 8 X 8, resulting in a total of 9,600 grids. Nine anchors of different scales and aspect ratios are generated for each grid (86,400 anchors).

We trained our network using the Tensorflow API[11] on a machine with an Intel Xeon 3.40GHz processor, 128GB RAM, and a GeForce GTX 1080 GPU with 8 GB of memory for about 12 hours.

3.3 Key-UI Screen Selection

Once our model has been trained in the previous step, it can be applied to *unseen* mobile development screencasts from YouTube to detect and extract UI screens from a video tutorial. However, the list of extracted UI screens can still contain duplicates. While duplicate frames were initially removed in the Distinct Frame Detection step of our approach (see Section 3.1.2), the focus there was on the full frames, rather than extracted UI screens. Two frames that contain identical UI screens may overall look dissimilar due to the fact that the rest of the frames besides the UI screen is dissimilar. Since the UI screens appear often only as a part of the image, the two frames would be considered overall distinct and both would be kept, leading to two identical UI screens being extracted. The additional filtering step we perform in this step assures that these cases are found and duplicate UI screens are removed.

Our Key-UI screen selection process can be described as follows. Given the set of extracted UI screens in video V as U = $\{U_1, U_2, \ldots, U_n\}$, this set might contain duplicate UI screens. To remove the duplicates, we could follow two approaches. The first involves comparing each two consecutive screens U_i and U_{i+1} to find if they are very similar. The problem when using this approach is the fact that there could be two non-consecutive UI screens such that U_i is a duplicate of U_j where $j \geq i+1$. Therefore, comparing only consecutive UI screens would result in this pair of duplicates not being detected. The second approach, which overcomes this problem, involves comparing all pairs of any two UI screens extracted, such that all duplicates can be located and removed. We used this second approach since it is more comprehensive.

[11] https://github.com/tensorflow/tensorflow

Our approach is deployed in a publicly available web-based tool[12]. The tool mines mobile programming screencasts, extracts the UI screens embedded in the videos, removes the duplicates, and then displays the resulting UI overviews of the screencasts to the user. By using the UIScreens tool, developers can, therefore, get an up-front overview of what the video is about without having to watch the entire video. An overview of the tool can be seen in Figure 3.

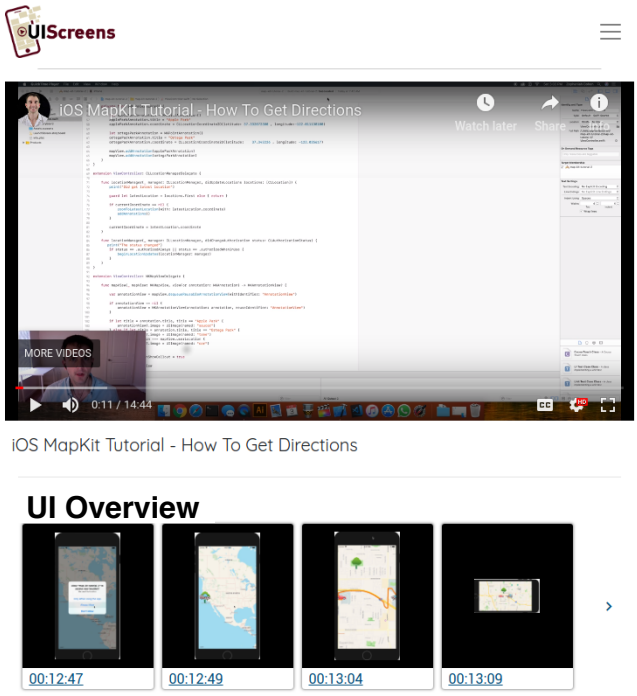


Figure 3: UIScreens tool

4 EMPIRICAL EVALUATION

We performed two empirical studies to evaluate our approach. The first study focuses on the accuracy of UIScreens in correctly identifying and extracting UI screens from mobile programming screencasts. The second study is a user study where 25 developers and computer science students with mobile programming experience were asked to evaluate the quality of the UI screens extraction, as well as the usefulness of extracting them. In the following subsections, we describe the research questions we address and the results of the studies. We release our complete dataset, results, and source code in our replication package[13].

4.1 Study I: Classification and Localization of UI screens

Motivation: In this study, we focus on evaluating our trained ***classification*** and ***localization*** model (step 2 of our UIScreens approach in Figure 2). Specifically, we aim to determine how well the model can classify video frames into UI and NonUI, and then locate the UI screens embedded in the UI frames. An accurate approach would precisely locate the UI screens, allowing us to then crop them correctly and present them to developers in the *UI overviews* of the videos in step 3 of our approach (see Figure 2). Therefore, our research question in this study is as follows:

- **RQ$_1$**: *How accurate is our approach at **classifying UI frames** and **locating UI screens** in mobile programming screencasts?*

Methodology: We trained our approach as described in Section 3 on the 4,000 UI and NonUI frames and their locations. We performed 10-fold cross-validation, where we split the data in each fold into training, validation, and testing sets. The validation and testing sets are each 10% of the data, while the training set is represented by the remaining 80% of the dataset. The validation dataset is used during the training to avoid overfitting. Our training process involved 4,000 iterations, until the network stopped improving and the validation loss was stable (*i.e.*, convergence occurred).

Evaluation metrics: To answer RQ_1, we used different metrics for each of the *classification* and *localization* tasks.

First, to evaluate the *binary classification* of the video frames into UI and NonUI, we used the standard metrics of *Precision*, *Recall*, F_1 *score*, and *Accuracy*. We denote T_p as the number of true positives, T_n as the number of true negatives, F_p as the number of false positives, and F_n as the number of false negatives. *Precision* is then defined as $P = \frac{T_p}{T_p+F_p}$ and *Recall* is computed as $R = \frac{T_p}{T_p+F_n}$. The F_1 *score* is the harmonic mean of precision and recall, defined as $F_1 = 2 \cdot \frac{P \cdot R}{P+R}$. Accuracy represents the percentage of correctly classified instances and is formally defined as $Acc = \frac{T_p+T_n}{T_p+T_n+F_p+F_n}$.

To evaluate the task of *locating UI screens* within the UI frames, we used the Intersection over Union (IoU) metric between the predicted location and the ground truth location. IoU is the standard metric in the object detection field, where it has been used in several competitions, such as PASCAL VOC Challenge [15], ImageNet Large Scale Visual Recognition Challenge [52], and Microsoft COCO [34]. Formally, IoU_{gt}^{pred} is defined as $IoU_{gt}^{pred} = \frac{Area\ of\ (pred \cap gt)}{Area\ of\ (pred \cup gt)}$, where *pred* is the predicted location and *gt* is the location of the ground truth. In other words, IoU measures the accuracy of the predicted area by comparing it to the ground truth area. Here, the ground truth is the annotated UI bounding box described in Section 3.1.3 and the predicted bounding box is the output of our trained model depicted in step 2 in Figure 2. The overall prediction performance is finally determined using an IoU threshold. More precisely, if the IoU is above a predefined threshold, the prediction is considered correct and if the IoU is below that threshold the prediction is considered incorrect. A lower IoU threshold typically results in an overall higher performance, while a higher IoU threshold generally decreases the overall performance. To account for this, we computed our results at different IoU thresholds between 0.70 and 0.90, with a step size of 0.10. This allows us to get a comprehensive view of the impact of the IoU threshold on the performance of our model. We then compute the *Average Precision (AP)* of our model at different IoU thresholds. AP has been used as a standard metric in several object detection competitions [11, 53]. We also computed the overall *Accuracy* of predicting the UI screens in our dataset.

[12]http://uiscreens.ddns.net/
[13]https://zenodo.org/record/3743842

Table 2: The classification results of the UI and NonUI Categories

Category	Precision	Recall	F-Score	Accuracy
UI	0.99	0.98	0.98	0.98
NonUI	0.98	0.99	0.98	0.98

Table 3: The average precision and accuracy of localizing UI screens using 10-fold cross-validation

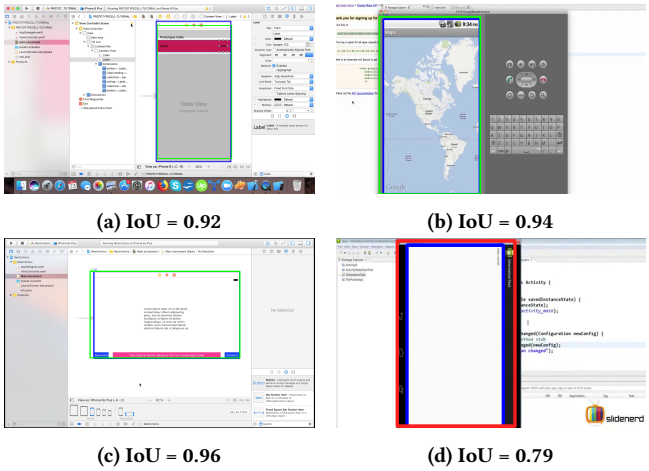
IoU Threshold	Average Precision	Accuracy
0.70	0.98	0.98
0.80	0.97	0.97
0.90	0.92	0.94

Results: Table 2 shows the results for the binary classification task using 10-fold cross-validation. Overall, our model performed extremely well, achieving both an *F-Score* and an *Accuracy* of 98% for both the UI and NonUI categories, as well as *Precision* and *Recall* of up to 99%. Table 3 presents the results of localizing the UI screens within UI frames with respect to the ground truth bounding box. The table shows the results at different IoU thresholds starting from 0.70 up to 0.90 with a step size of 0.10. Our model achieved an *AP* of 92% at an IoU threshold of 0.90, and an *Accuracy* of 94% at the same threshold. At lower IoU values, both the *AP* and the *Accuracy* increase. However, since we want the prediction to be as precise as possible, we consider the IoU threshold of 0.90 to be the most adequate for an overall picture of the results.

Figure 4 depicts a few examples of UI screen predictions made by our model compared to the ground truth bounding boxes. As it can be clearly seen in Figures 4a, 4b, 4c, the predicted UI screen location almost overlaps with the ground truth bounding box, yielding a high localization accuracy. In these examples, the IoU threshold was set to 0.90, therefore, 4d was considered an incorrect prediction. The speed of our model's prediction is 1.7 seconds per frame.

4.2 Study II: UIScreens Evaluation by Developers

Motivation: The second part of our evaluation is represented by a user study which focuses on assessing the end result of UIScreens, namely the ***UI overview*** generated at the end of the last step in our approach (see Figure 2). We believe that *high-quality UI overviews* have the potential to help developers get a quick comprehension of the main points of a program explained in a video, which could save them time when searching for helpful videos for their information needs. Therefore, in this user study, we aimed to evaluate both the *quality* of the UI overviews generated by our approach, as well as their perceived ***usefulness*** by developers. In terms of *quality*, we specifically focused on two aspects relating to the UI screens extracted by our approach. The first aspect is the UI screens' ***uniqueness***: the extracted UI overview should not contain duplicate or very similar UI screens, in order to avoid overwhelming the developer with screens that do not convey new information.



(a) IoU = 0.92 (b) IoU = 0.94

(c) IoU = 0.96 (d) IoU = 0.79

Figure 4: Detecting the UI bounding box and computing the Intersection over Union (IoU) (ground truth is in blue, correct predictions in green, and incorrect predictions in red)

The second quality aspect considered is the UI overview's ***sufficiency***: the extracted UI screens should be sufficient to offer a comprehensive overview of the video and its main points.

We summarize these goals in two research questions:

- **RQ**$_2$: *To what extent are the extracted UI screens considered **unique** and **sufficient** by developers?*
- **RQ**$_3$: *What are the perceived **benefits** of the UI overview extracted by UIScreens for developers?*

Methodology: The study was conducted through an online survey composed of two main sections. The first section was designed to capture demographic data, such as the main occupation (professional developer, academic, student) and the mobile programming experience (iOS and/or Android and number of years) of our participants. Each participant was required to have at least 6 months experience in at least one of the two mobile programming platforms to be qualified for participating in our survey. The study participants were recruited through announcements on professional social media channels.

For this study, we collected a *brand new set of 50 programming screencasts* (25 iOS and 25 Android), on which we applied our approach. This was done in order to avoid any bias that could be caused by applying UIScreens on a video from which frames were used during the training of our model. This is important for ensuring that our model is generalizable and that our evaluation is unbiased. The average length of the videos in this study was ~ 10 minutes. Each participant was assigned one iOS and one Android video. For each of the two videos, they were asked to first watch the video in its entirety and then evaluate the extracted UI screens by indicating their agreement level with two statements. The first statement referred to the *sufficiency* of the extracted UI screens ("The list of UI screens extracted is *sufficient* to understand what are the main concepts discussed in the video") and the second one referred to their uniqueness ("The list of UI screens extracted does not present duplicate information, *i.e.*, all UI screens presented are *unique*"). We ensured that the same video is not displayed to more

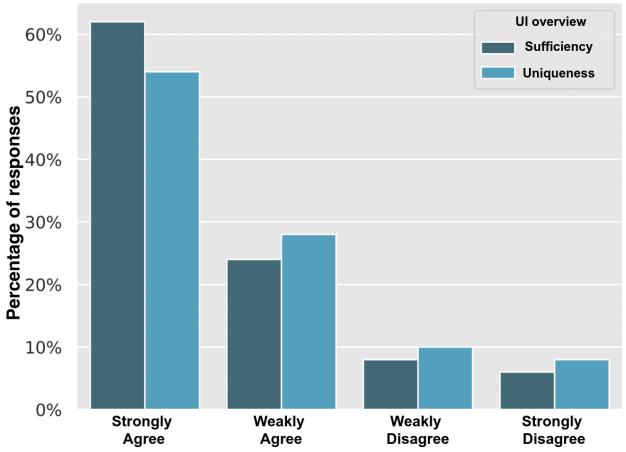


Figure 5: Quality of extracted UI overviews (sufficiency and uniqueness)

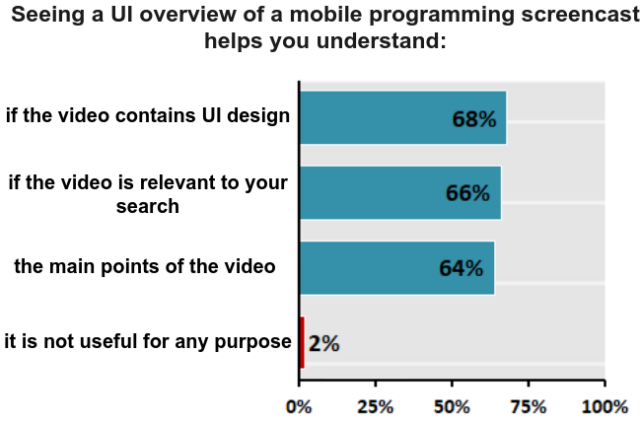


Figure 6: Usefulness of an UI overview

than one respondent by evenly selecting among the videos. The answers to these two questions were on a 4-point Likert scale, namely: Strongly Agree, Weakly Agree, Weakly Disagree, and Strongly Disagree. The questionnaire also contained a third question for each video, concerning the perceived *benefits of UI overviews* (RQ_2). A list of possible answers was displayed to each participant, who could select one or more of the available options (see Figure 6 for the list of possible answers) or enter their own answer. The last answer in Figure 6 ("It is not useful for any purpose") is exclusive (*i.e.*, all other choices would be automatically unselected when this choice is selected).

Results: A total of 25 developers completed our survey, having various levels of experience in Android and iOS development. Most of our participants were M.S. and Ph.D. students in computer science with 28% of the total participants in each of these categories. In addition, 24% of the participants were undergraduate students and the remaining 20% were professional developers.

Figure 5 shows the results of the study for RQ_2. In 85% of their responses, developers either weakly or strongly agreed that the UI screens extracted by our approach were sufficient in understanding what the main elements discussed in the video were. At the same time, 83% of responses either weakly or strongly agreed that the extracted UI screens were unique when compared to each other. This indicates that UIScreens can efficiently extract distinct and sufficient UI screens in order to provide comprehensive ***UI overviews*** for mobile programming screencasts.

Figure 6 depicts the answers to the question: "Seeing a UI overview of a mobile programming screencast helps you understand:..". Only 2% of the participants indicated that seeing a UI overview of a mobile programming screencast is not useful for any purpose. In total, 68% of the participants indicated that the extracted UI overview can help them understand if the video contains UI design or not, 66% thought the UI screens can help them understand the relevance of a video for their search needs, and 64% said the UI screens help them understand the main points of the video.

5 THREATS TO VALIDITY

The main threats to the *internal validity* of our findings relate to the subjective nature of the manual data classification and annotation tasks. To mitigate these threats, two authors participated in the classification and annotation of each frame and reached an agreement upon any conflict. For our second evaluation study, the threats to the internal validity of our findings relate to the possibility that our model overfit the training data. In order to mitigate this threat, we randomly selected a brand new set of 50 screencasts from YouTube and applied our approach on this new set of videos for the user study. The 50 videos were unseen to the model and have not been used during the training phase. In addition, in order to ensure that the study participants are not influenced by fatigue, we limited the number of videos they watched and answered questions about to two, and limited the maximum length of each video to 10 minutes.

Construct validity concerns the metrics used to evaluate our approach for classifying and locating UI screens. We mitigated this threat by using well-established performance measurements from the classification and object detection fields [11, 52, 53]. In addition, we reported results at different IoU thresholds for the localization task, in order to account for the variability in accuracy and average precision that can occur when different IoU thresholds are chosen.

Regarding the threats to *external validity*, our results may not be generalizable to all mobile programming screencasts. To mitigate this threat, we trained UIScreens on a diverse set of videos covering a variety of programming languages, IDEs, different OS platforms, different phone orientations, etc. In addition, our video dataset is the largest compared to other works on video programming tutorials [13, 41, 43, 44, 48, 65]. During our second study, we evaluated our model on unseen videos to ensure that it is generalizable.

6 RELATED WORK

In the following subsections, we survey a sizable body of recent works that aim to mine video programming tutorials, along with other works related to analyzing mobile UIs.

6.1 Analyzing Video Programming Tutorials

Programming screencasts have been analyzed by several works over the past few years due to the increase of their popularity among developers, who use them as a source of information to support their programming tasks. MacLeod *et al.* [36, 37] performed a set of interviews with programming screencast creators and revealed their motivations for creating screencasts. Bao *et al.* proposed two approaches that analyzed programming screencasts. The first one automatically produces time-series HCI data from development screen-captured videos [3], while the second one is meant to capture the workflow of programming tasks and display it along with the video [4]. Zhao *et al.* proposed ActionNet to also track the workflow of programming screencasts, using a CNN-based classifier with an Inception-Resnet V2 backbone [69].

Several works aimed to detect code frames in programming video tutorials [1, 43, 44, 47, 48] and to extract source code from them [28, 65]. Ott *et al.* proposed a CNN-based classifier to (i) classify video frames based on the presence or absence of source code in them [43], and to (ii) detect the programming language used in screencasts [44]. In our previous work, we located the code editing window in programming screencasts using object detection algorithms [1, 2]. A tool called CodeTube was proposed by Ponzanelli *et al.* [48] to fragment programming video tutorials into smaller parts and then categorize each fragment based on its intent (*e.g.*, code implementation fragment). CodeTube also identified source code in video frames and extracted it using OCR for indexing. Khandwala *et al.* proposed an approach to extract and combine different code snippets from screencasts and display them using the Codemotion tool [28]. Yadid and Yahav [65] introduced a tool to extract source code from video tutorials using OCR and a statistical language model.

Some previous works aimed to automatically tag software development video tutorials [14, 45]. In addition, Poché *et al.* [46] proposed a YouTube comment classification approach based on machine learning techniques. Moreover, Moslehi *et al.* [41] proposed an approach to link source code files to the corresponding screencasts by leveraging UI information, as well as the audio transcripts.

Our goal is different than all these previous works since we are targeting the extraction of UI screens in order to provide *UI overviews* of mobile programming screencasts. To the best of our knowledge, this is the first work that has aimed to extract mobile UI screens from videos.

6.2 Analyzing Mobile Graphical User Interfaces

There have been several works that analyzed the changes made to the GUI during the evolution of mobile apps, with the purpose of documenting them. Some approaches addressed this problem by automatically detecting the changes to the GUI [64] and summarizing them [40]. Moran *et al.* [40] extract a set of GUI images and their metadata files from different versions of an app and compare them using computer vision algorithms. The main goal of our work is different, as we focus on screencasts.

Initially, mock-ups of UI screens can be designed using several prototyping techniques [10, 30]. Developers then use this prototype to design the actual UI screens and integrate them into the mobile app implementation. The implementation of the GUI, however, could differ from the mock-ups. Moran *et al.* [39] focus on detecting violations of the GUI design. Similarly, other work focuses on validating the GUI consistency across different platforms [16, 23]. A recent work was also proposed by Moran *et al.* [38] to detect and classify different components of the mock-up design.

A notable approach was also proposed by Bernal-Cardenas *et al.* [8] to enable searching for mobile app UI screens through a light-weight search engine tool (Guigle). The tool currently indexes a corpus of UI screens extracted from the Google play store. The users can type a query as input, and a list of UI screens along with the app name is returned. Our approach is different as it identifies and extracts the most representative UI screens from mobile programming screencasts. While Guigle can help developers with the conceptualization of a GUI, developers still have to implement them. Our approach could help with this latter aspect and could potentially be integrated with Guigle to further support developers in going from concept to implementation.

Another approach, called GUIFetch, was proposed by Behrang *et al.* [7] and it enables using a UI sketch as input to search for code that may help in implementing the UI.

7 ACKNOWLEDGMENTS

Sonia Haiduc was supported in part by the National Science Foundation grants CCF-1846142 and CCF-1644285.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a novel approach, called UIScreens, to locate and extract UI screens embedded in mobile programming screencasts, with the purpose of offering developers a UI-focused overview of a video. Our approach extracts the deep features from video frames using a CNN and these features are then fed into an object detector to identify the exact location of UI screens within the video frames. UIScreens was also implemented as a freely available tool. We conducted two evaluation studies to assess our approach in terms of its accuracy, and the quality and usefulness of its results. The evaluation showed that UIScreens was able to accurately classify and locate UIs in video frames. Additionally, UIScreens received positive feedback from mobile developers, showing potential for our approach to help developers in navigating and understanding the contents of mobile programming screencasts. To our knowledge, UIScreens is the first approach to perform UI screen extraction from mobile development screencasts.

In our future work, we plan to explore and identify in more detail the particular GUI elements present in the extracted UI screens and to allow indexing and searching based on UI element types (*e.g.*, drop-down list). Furthermore, we plan to conduct more user studies where professional developers and computer science students get to evaluate our tool in the context of specific program comprehension and UI programming tasks. We also plan to explore UI extraction from other types of programming videos besides those focused on mobile applications.

REFERENCES

[1] Mohammad Alahmadi, Jonathan Hassel, Biswas Parajuli, Sonia Haiduc, and Piyush Kumar. 2018. Accurately predicting the location of code fragments in programming video tutorials using deep learning. In *Proceedings of the 14th*

International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE'18. ACM Press, Oulu, Finland, 2–11.

[2] Mohammad Alahmadi, Abdulkarim Khormi, Biswas Parajuli, Jonathan Hassel, Sonia Haiduc, and Piyush Kumar. 2020. Code Localization in Programming Screencasts. *Empirical Software Engineering* (2020), 1–37.

[3] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, Xin Xia, and Bo Zhou. 2017. Extracting and analyzing time-series HCI data from screen-captured task videos. *Empirical Software Engineering* 22, 1 (2017), 134–174.

[4] Lingfeng Bao, Zhenchang Xing, Xin Xia, and David Lo. 2018. VT-Revolution: Interactive programming video tutorial authoring and watching system. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2802916

[5] Patti Bao, Jeffrey Pierce, Stephen Whittaker, and Shumin Zhai. 2011. Smart phone use by non-mobile business users. In *Proceedings of the 13th international conference on human computer interaction with mobile devices and services*. ACM, 445–454.

[6] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. 2006. Surf: Speeded up robust features. In *European conference on computer vision*. Springer, 404–417.

[7] Farnaz Behrang, Steven P Reiss, and Alessandro Orso. 2018. GUIfetch: Supporting app design and development through GUI search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 236–246.

[8] Carlos Bernal-Cárdenas, Kevin Moran, Michele Tufano, Zichang Liu, Linyong Nan, Zhehan Shi, and Denys Poshyvanyk. 2019. Guigle: a GUI search engine for Android apps. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 71–74.

[9] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1589–1598. https://doi.org/10.1145/1518701.1518944

[10] Tiago Silva Da Silva, Angela Martin, Frank Maurer, and Milene Silveira. 2011. User-centered design and agile methods: A systematic review. In *Agile Conference (AGILE), 2011*. IEEE, 77–86.

[11] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. 2016. R-FCN: Object detection via region-based fully convolutional networks. *arXiv:1605.06409 [cs]* (May 2016). http://arxiv.org/abs/1605.06409 arXiv: 1605.06409.

[12] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2012. Using IR Methods for Labeling Source Code Artifacts: Is It Worthwhile?. In *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC'12)*. IEEE, Passau, Germany, 193–202.

[13] Mathias Ellmann, Alexander Oeser, Davide Fucci, and Walid Maalej. 2017. Find, understand, and extend development screencasts on YouTube. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*. ACM, 1–7.

[14] Javier Escobar-Avila, Esteban Parra, and Sonia Haiduc. 2017. Text retrieval-based tagging of software engineering video tutorials. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*. IEEE, Buenos Aires, Argentina, 341–343.

[15] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. 2010. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision* 88, 2 (2010), 303–338.

[16] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 308–318.

[17] Leopoldina Fortunati and Sakari Taipale. 2014. The advanced use of mobile phones in five European countries. *The British journal of sociology* 65, 2 (2014), 317–337.

[18] Ross Girshick. 2015. Fast R-CNN. *arXiv:1504.08083 [cs]* (April 2015). http://arxiv.org/abs/1504.08083 arXiv: 1504.08083.

[19] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2013. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv:1311.2524 [cs]* (Nov. 2013). http://arxiv.org/abs/1311.2524 arXiv: 1311.2524.

[20] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 24th ACM/SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, Seattle, WA, USA.

[21] Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* 21, 9 (2009), 1263–1284.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *arXiv:1512.03385 [cs]* (Dec. 2015). http://arxiv.org/abs/1512.03385 arXiv: 1512.03385.

[23] Andreas Holzinger, Peter Treitler, and Wolfgang Slany. 2012. Making apps useable on multiple different mobile platforms: On interoperability for business application development on smartphones. In *International Conference on Availability, Reliability, and Security*. Springer, 176–189.

[24] Wei Hu, Yangyu Huang, Wei Li, Fan Zhang, and Hengchao Li. 2015. Deep convolutional neural networks for hyperspectral image classification. *J. Sensors* 2015 (2015), 258619–258619. https://doi.org/10.1155/2015/258619

[25] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. 2017. Speed/accuracy trade-offs for modern convolutional object detectors. In *IEEE CVPR*, Vol. 4.

[26] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167 [cs]* (Feb. 2015). http://arxiv.org/abs/1502.03167 arXiv: 1502.03167.

[27] Amy K Karlson, Brian R Meyers, Andy Jacobs, Paul Johns, and Shaun K Kane. 2009. Working overtime: Patterns of smartphone and PC usage in the day of an information worker. In *International Conference on Pervasive Computing*. Springer, 398–405.

[28] Kandarp Khandwala and Philip J. Guo. 2018. Codemotion: Expanding the design space of learner interactions with computer programming tutorial videos. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale - L@S '18*. ACM Press, London, United Kingdom, 1–10. https://doi.org/10.1145/3231644.3231652

[29] 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.

[30] Kati Kuusinen and Tommi Mikkonen. 2013. Designing user experience for mobile apps: Long-term product owner perspective. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, Vol. 1. IEEE, 535–540.

[31] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, Lincoln, Nebraska, USA, 476–481.

[32] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. 1999. Object recognition with gradient-based learning. In *Shape, Contour and Grouping in Computer Vision*. Springer-Verlag, London, UK, UK, 319–345. http://dl.acm.org/citation.cfm?id=646469.691875

[33] Stefan Leutenegger, Margarita Chli, and Roland Siegwart. 2011. BRISK: Binary robust invariant scalable keypoints. In *2011 IEEE international conference on computer vision (ICCV)*. Ieee, 2548–2555.

[34] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2014. Microsoft COCO: Common objects in context. *arXiv:1405.0312 [cs]* (May 2014). http://arxiv.org/abs/1405.0312 arXiv: 1405.0312.

[35] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. In *Proceedings of the 15th ACM/SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'09)*. ACM, Paris, France, 557–566.

[36] Laura MacLeod, Andreas Bergen, and Margaret-Anne Storey. 2017. Documenting and sharing software knowledge using screencasts. *Empirical Software Engineering* 22, 3 (June 2017), 1478–1507. https://doi.org/10.1007/s10664-017-9501-9

[37] Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. 2015. Code, camera, action: How software developers document and share program knowledge using YouTube. In *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC'15)*. Florence, Italy, 104–114.

[38] Kevin 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* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2844788

[39] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. *arXiv preprint arXiv:1802.04732* (2018).

[40] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. 2018. Detecting and summarizing GUI changes in evolving mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 543–553.

[41] Parisa Moslehi, Bram Adams, and Juergen Rilling. 2018. Feature location using crowd-based screencasts. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. ACM Press, Gothenburg, Sweden, 192–202. https://doi.org/10.1145/3196398.3196439

[42] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 248–259.

[43] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. A deep learning approach to identifying source code in images and video. In *Proceedings of the 15th IEEE/ACM Working Conference on Mining Software Repositories*. 376–386.

[44] Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, and Erik Linstead. 2018. Learning lexical features of programming languages from imagery using convolutional neural networks. , 336-339 pages.

[45] Esteban Parra, Javier Escobar-Avila, and Sonia Haiduc. 2018. Automatic tag recommendation for software development video tutorials. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 222–232.

[46] Elizabeth Poché, Nishant Jha, Grant Williams, Jazmine Staten, Miles Vesper, and Anas Mahmoud. 2017. Analyzing user comments on YouTube coding tutorial videos. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 196–206.

[47] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. CodeTube: Extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE'16)*. ACM, Austin, TX, 645–648.

[48] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Rocco Oliveto, Massimiliano Di Penta, Sonia Cristina Haiduc, Barbara Russo, and Michele Lanza. 2017. Automatic identification and classification of software development video tutorial fragments. *IEEE Transactions on Software Engineering* (2017). https://doi.org/10.1109/TSE.2017.2779479

[49] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural Networks* 12, 1 (1999), 145–151.

[50] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2010. Summarizing Software Artifacts: A Case Study of Bug Reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. ACM, Cape Town, South Africa, 505–514.

[51] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. *arXiv:1506.01497 [cs]* (June 2015). http://arxiv.org/abs/1506.01497 arXiv: 1506.01497.

[52] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[53] Abhinav Shrivastava and Abhinav Gupta. 2016. Contextual priming and feedback for faster r-cnn. In *European Conference on Computer Vision*. Springer, 330–348.

[54] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556 [cs]* (Sept. 2014). http://arxiv.org/abs/1409.1556 arXiv: 1409.1556.

[55] Clayton Stanley and Michael Byrne. 2013. Predicting Tags for StackOverflow Posts. In *Proceedings of 12th International Conference on Cognitive Modelling (ICCM'13)*. Carleton, CA, 414–419.

[56] Statista. 2018. Number of smartphone users worldwide from 2014 to 2020 (in billions). (2018). https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide

[57] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. 2016. Inception-v4, Inception-ResNet and the impact of residual connections on learning. *arXiv:1602.07261 [cs]* (Feb. 2016). http://arxiv.org/abs/1602.07261 arXiv: 1602.07261.

[58] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the inception architecture for computer vision. *arXiv:1512.00567 [cs]* (Dec. 2015). http://arxiv.org/abs/1512.00567 arXiv: 1512.00567.

[59] Kai Tian, Meghan Revelle, and Denys Poshyvanyk. 2009. Using Latent Dirichlet Allocation for Automatic Categorization of Software. In *Proceedings of the 6th IEEE Working Conference on Mining Software Repositories (MSR'09)*. IEEE, Vancouver, Canada, 163–166.

[60] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. 2013. Selective search for object recognition. *International Journal of Computer Vision* 104, 2 (2013), 154–171.

[61] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. ACM, Singapore, Singapore, 87–98.

[62] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR'15)*. IEEE, Florence, Italy.

[63] Chunlei Xia, Longwen Fu, Hui Liu, and Lingxin Chen. 2018. In situ sea cucumber detection based on deep learning approach. In *2018 OCEANS-MTS/IEEE Kobe Techno-Oceans (OTO)*. IEEE, 1–4.

[64] Qing Xie, Mark Grechanik, Chen Fu, and Chad Cumby. 2009. Guide: A GUI differentiator. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 395–396.

[65] Shir Yadid and Eran Yahav. 2016. Extracting code from programming tutorial videos. In *Proceedings of the 6th ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'16)*. ACM, Amsterdam, The Netherlands, 98–111.

[66] Jian Yang, Yang Xiao, Zhiwen Fang, Naiwen Zhang, Li Wang, and Tao Li. 2017. An object detection and tracking system for unmanned surface vehicles. In *Target and Background Signatures III*, Vol. 10432. International Society for Optics and Photonics, 104320R.

[67] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.

[68] Yue Zhang, Bin Song, Xiaojiang Du, and Mohsen Guizani. 2018. Vehicle tracking using surveillance with multimodal data fusion. *IEEE Transactions on Intelligent Transportation Systems* 19, 7 (2018), 2353–2361.

[69] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, Guoqiang Li, and Shanghai Jiao Tong. 2019. ActionNet: Vision-based workflow action recognition from programming screencasts. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE'19)*.