International Journal of Pure and Applied Mathematics
Volume 118 No. 24 2018
ISSN: 1314-3395 (on-line version)
url: http://www.acadpubl.eu/hub/
Special Issue

http://www.acadpubl.eu/hub/

Extraction and Classification of User Interface Components from an Image

Saad Hassan[1], Manan Arya[2],
Ujjwal Bhardwaj[3],Silica Kole[4]
Computer Science and Engineering,
Bharati Vidyapeeths College of Engineering,
New Delhi, India
hassan.saad.mail@gmail.com
mananarya22@gmail.com
ujjwalb1996@gmail.com
silica.kole@bharatividyapeeth.edu

May 25, 2018

Abstract

Implementing the Graphical User Interface (GUI) after creating mockups design, which generally reflects the design choices for layouts, colours, typography, visual components of the product, for the website or mobile application is very time consuming process. This steals the part of developers time which he would have dedicated on the logic and the actual functionality of the application. Also, the developer has to produce separate code to produce the GUI for Web based applications, Android native applications, the IOS applications and Windows applications. In this paper, we describe the approach using which we can detect the GUI components from a mockup using computer vision and deep convolutional network which can be directly used to further produce the code of GUI for multiple platforms at once.

Key Words:User Interface Components Detection, Computer Vision, Machine Learning, Pattern Recognition, Deep Convolutional Network.

1 Introduction

Development of the Graphical User Interface (GUI) of a software application is generally broken into several parts. The work is distributed among the Designers and the Front-end Developers. A designer prepares the layout interface which is then converted to a design interface. This design interface is then handed over to the Frontend- developer who implements the interface in code followed by implementing the front end logic. We know that implementing the GUI after creating mockups is very time consuming and prevent the developers from dedicating the majority of their time on the actual functionality and logic of the application. This decreases the efficiency of the developer to produce actual logic of the software. The layout interface created by the designer contains a lot of essential information which can be extracted and used in further development processes including automation of various tasks.

In this paper, we describe our approach using which we can detect the GUI components from a mockup using computer vision and deep convolutional network, which can be directly used to further produce the code of GUI for multiple platforms at once. We are trying to generate a standard hierarchical format which can be further used as mapping to generate platform dependent codes.

2 BACKGROUND AND RELATED WORK

Developers have always been trying to generate applications that could help the people to automate tasks for their day to day work. However, development of tools to help developers has only been addressed quite recently. Automatically generating the code for programs using various machine learning techniques is a relatively new field of research. The work by Gaunt et al. [1] has made it possible to generate the source code by learning the relationship between input and output by differentiable interpreters. Ling et al. [2] have demonstrated the synthesis of a program from

structured program specification and a mixed natural language as input.

Although the generation of computer programs is an active research field now, the generation of programs from the visual inputs or images is still a nearly unexplored field. The closest related work is a method developed by Tony Beltramelli [3] who has demonstrated how deep learning methods can be leveraged to train an end-to-end model that generates GUI code from input mockup images with over 77% accuracy for mobile and web-based technologies. But it is important to note that context of components and their styles were not satisfactorily obtained as output in Tonys Work. However, the hierarchical structure of the GUI elements was preserved in his work. It is also pertinent to mention that the model was trained on a limited dataset which is not the case with our implementation

3 IMPLEMENTATION

In this section, we describe the approach to detect components and classify them in the GUI of various platforms, including mobile and web-based technologies. We have selected screenshot of a minimal mobile interface based on the material design. Even though it limits our approach to mobile-based applications, the approach is expected to work with a similar efficiency in web-based applications as well.

A. Text Region Detection
B. Masking the Detected Text Regions
C. Component Detection
D. Segmenting the Available Contours and Passing to Classifier
E. Training Model with UI Elements
F. Storing Meta-information of Extracted Components

A. Text Region Detection

We start our approach by detecting regions of text first so that we can mask the area with dominant background or remove these

regions from other detected contours later on in the process. We learned from various trials that certain unwanted contours are detected in an image containing text regions while detecting shapes. Such commotions are bound to hinder processes in our model and hence are masked.

The previous work on detection of text from images is quite a lot and commendable, we tried to leverage on these works in order to get the regions or as you may say the bounding rectangles for each text blocks. [4, 5, 6, 7] suggests methods for detecting text in region, scenic as well as texture based methods. The limitation of using the above methods in our case is the complexity and computational time required.

Considering our case to lower detection time and computational complexity, we try to apply canny edge detector [8] to an image. Fig.1. shows the GroundTruth GUI image on which we have applied the text detection and Fig.2 shows the transformed image after applying canny edge detection. This helps to generate white pixels where there are edges in the image. In order to remove the outlines that are not needed and arent part of the text, we apply the median blur filter to the above image. Fig.3. shows the transformed GUI image after applying the median blur. Now with most of the borderline eliminated, we have with us the text region pixels. What we require now is to find the connected text regions and segment regions that are not part of them. To do this we apply increasingly aggressive dilation, taking a matrix of 5x5 as a kernel. Fig.4. shows the transformed GUI image after applying the dilation. Applying contour detection on this result we are able to find the required text region and its bounding rectangles. Fig.5. shows the contours detected. The bounding rectangle coordinates will later help us masking the text regions. Though the above method has limitations over noisy and scenic images, we ignore that case as normally GUI Mockups aren't noisy or with a scenic background. However, in need of a greater accuracy, one can accommodate [9, 10, and 11] in a trade-off computational complexity and time.

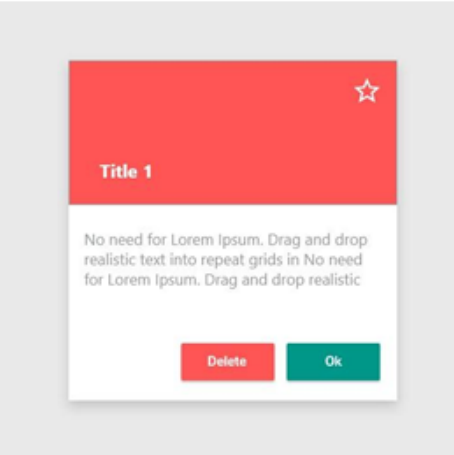

Fig. 1. GroundTruth GUI

Fig. 2. GUI after canny edge detection

Fig. 3. GUI after median blur

Fig. 4. GUI after dilation

B. Masking the detected Text Regions

After the text regions are detected, we need to mask them in order to avoid all the unwanted contours that may be realized from these text regions, further creating problems in the detection of the components. This is done by detecting the most relevant colour around the text or of the component holding that text. We identify

this by ranking the colours in order of their occurrences and if any colour has occurrence greater 60-70% we select that as our masking colour, else we average top 3 colours for the mask. Averaging helps to detect colour for gradient background. Fig 6. shows image after the GUI Text Detection and Fig 7. Shows the image of the GUI after masking the detected text regions.

Fig. 5. GUI after Contour Detection

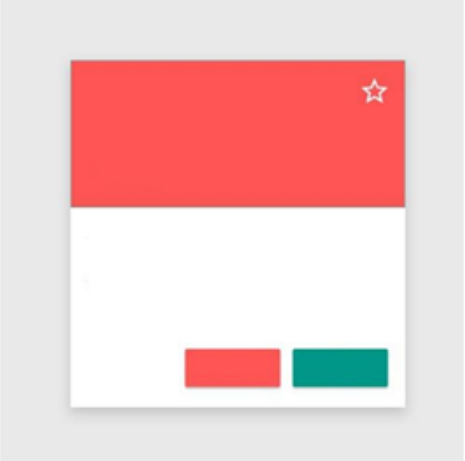

Fig. 6. GUI after Masking the detected text regions

C. Components Detection

The components in the User Interface of the aforementioned platforms generally include Buttons, Text Views, and Input Texts etc. These components are identified using Contour Finding [12] techniques. The Component Detection step serves as the initial step of our Component Extraction Model. The basic flow is to first detect the outline of the shapes present in our image and then find out the centre of contours. Pre-processing of the images has to be done before detecting the outlines. We follow this by first loading and resizing the image and then decreasing its size so that the shape of components can be approximated better. Then we convert the image to grayscale, blur it and invert the colours. Blurring the image using the Gaussian blur helps in reducing high-frequency noise. Binarization is done using the edge detection and thresholding the image. Threshold Images are also known as black-and-white. [13] Thresholding of the image helps in classifying the image intensities in one or more classes or labels of various shades of grey, which reduces the noise in the image to a minimum.

After the image is transformed and threshold, we find out the contours of the components inside the image. Besides this, we compute the centroid for each contour. A certain lower bound is applied to the area of detected contours to limit the detection such that certain irrelevant detections are ignored.

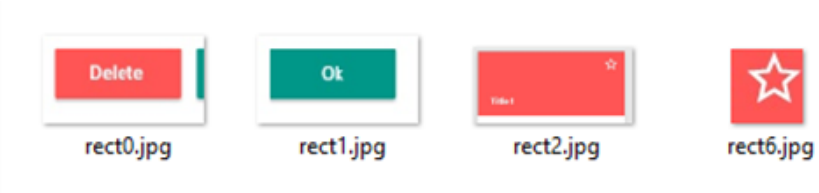

Fig. 7. Components Segmented after contour detection

D. Segmenting the Available Contours and Passing to Classifier

After the contour of the components are detected in the previous step, we identify a rectangular region bounding the component. The bounding rectangle with a constant padding is used to crop the components from the image. Here we used a

constant padding of 16px around the bounding rectangle. Since the image is in the form of a two dimensional array, we can extract a portion of the array matrix of the original image and save it in a separate variable to obtain the cropped component. The cropped component image is then saved as a new file and sent to the component predictor to identify the type of UI component using the extracted properties.

E. Training Model with UI Elements

Our primary aim is to extract the components from the GUI screenshots such that they can themselves be classified and then further be used for classification. For each component, we predict its likeness to one of the labelled classes of components in the trained model by incorporating the classification model learned using previous inputted images along with the images used to prepare the initial model.

The transfer learning method has been used with a model that has already been trained on large datasets. We simply retrain that model with our datasets and we are good to go for classification. We did this so because training deep convolutional neural networks from scratch can take too long and requires quite good computation power. But transfer learning can help us achieve approximately the same result in a shorter period.

In our case, we are using a model with MobileNet [14] architecture trained on ImageNet Challenge Large dataset. The architecture to choose comes with tradeoff like speed, accuracy and size of the dataset. Comparing InceptionV3 with MobileNet, InceptionV3 provides better accuracy than MobileNet on ImageNet but requires more processing steps than the MobileNet which is 1/4 of its size.

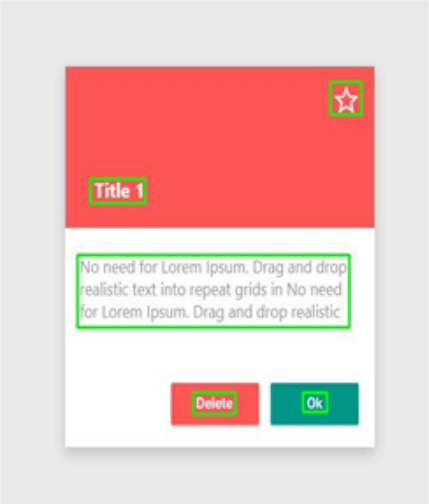

Fig. 8. GroundTruth GUI with detected components.


```
{
    "components": [
        {
            "label": "Header",
            "dimensions": {
                "x": 75,
                "y": 65,
                "width": 362,
                "height": 157
            },
            "primaryColor": "#ff5555",
            "components": [
                {
                    "label": "star icon",
                    "dimensions": {
                        "x": 391,
                        "y": 93,
                        "width": 8,
                        "height": 9
                    },
                    "primaryColor": "#ffffff"
                },
                {
                    "label": "textview"
```


Fig. 9. Hierarchal format in the form of JSON.

We retrain the model with UI elements like Buttons (basic, primary, success, danger, info, and link), Inputs, and Checkboxes, Radio, Select components and Icons. For Icons, we gathered

names for each basic icon from Googles Open Source Material Design Icons Library. For each icon type, we gathered 50-80 images and cleansed data for better accuracy. The input images are initially re-sized to 224 pixels configuration for MobileNet. After the model is trained we are ready for the classification of the components cropped in the previous step.

F. Storing Meta-information of Extracted Components

The Retrained Model is used to classify the components and label it accordingly. The label along with bounding rectangle coordinates and extracted features like primary and secondary colours is stored for each component in a hierarchical format. We chose to use JSON (JavaScript Object Notation) as the standard for storing these details. The easy transformation and being widely used in Web and Mobile technologies JSON could help in further experimenting with the Meta information identified from extracted components. This hierarchical format as shown in Fig. 9 can then be used by a language mapping model which could generate the code for each platform independently.

4 EXPERIMENTS AND RESULTS

In this section, experimental setup details have been explained along with the result generated from it.

A. Training and Validation on the UI Dataset

Since we crop and extract components from the GUI screenshots before classifying them using our trained model, our Dataset consist only of isolated icons and buttons image. To the best of our knowledge, no dataset containing icons images with their label existed at the time this paper was written. Hence, we prepared our own dataset, storing similar icons in a single directory with their classified label as the directory name. Our dataset was limited to a fixed number of chosen icons and buttons and hence the system is expected to fail in identifying other icons unless trained accordingly.

The model was trained for 10k training steps, at each training

steps 10 images were taken at random from training datasets, the bottlenecks of each image was fed to final CNN layer to get the predictions. The result is then compared to actual label and therefore, accuracy is found and improved at each step. Greater the steps of training, data get greater accuracy. The true performance measure is validation accuracy. The network is overfitting if the training accuracy is high and we train the model with huge dataset but do not test it much. In this case, the neural network is memorizing features particular to the training dataset images, that don't help it in classifying images and the validation accuracy remains low. In our case, we got the validation accuracy of 90.2% which provided decent results predicting the UI elements. The below figures (Fig.10 and Fig.11) respectively shows the accuracy graph and cross entropy graph of our trained model and dataset.

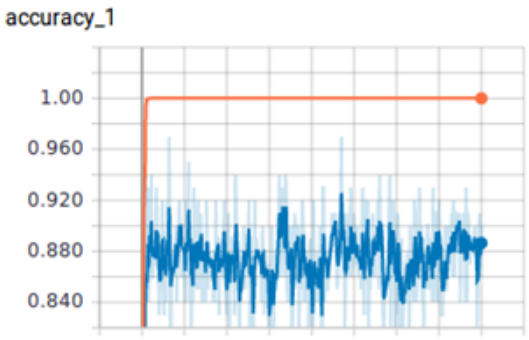

Fig. 10. Accuracy Graph.

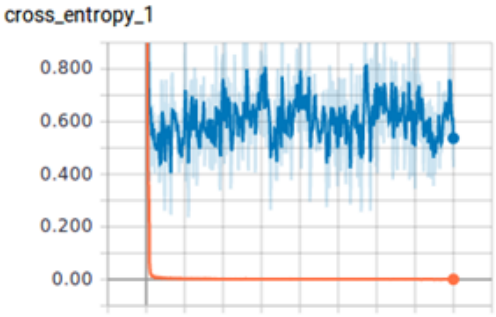

Fig. 11. Cross Entropy Graph

5 LIMITATIONS

In this section, we describe a few factors that may lead to certain unsatisfactory outcomes. Firstly, while identifying the text regions there may be some cases when icons are falsely identified as text. In these cases, either we take regions with an area greater than some constants or we apply more robust text detection on the image. Scenic background along with gradients can also cause trouble detection some portion of text. While our focus was to mainly explore what is possible we ignored this limitation for future optimizations.

Secondly, while detecting contours in the input images, some essential details like shadows, gradient and images were not preserved. Apply different layers of detection can help solving these problems. Also applying deep neural networks for background overlays and shadows can help in getting modern UI hierarchy right.

These limitations can be overcome by applying the work that has already been done tuning the characteristics and properties of filters. The impressive outcomes for text detection and contour detection by a number of researches can help in evolving our work for better accuracy.

6 CONCLUSIONS

With this paper, we presented a novel approach to extract and classify the various components from GUI screenshots of different platforms like web and mobile based technologies. Our model demonstrates the potential of being an integral part of a system that intends to optimize the task of developing a Graphical User Interface by automating the process. With the use of limited parameters and a relatively small dataset, we have only scratched the surface of a much-extended area. Implementing the Graphical User Interface after creating mockups is very time consuming and prevent the developers from dedicating the majority of their time on the actual functionality and the logic of the application. This work can therefore help in enabling developers to focus on the task that is more important like logic implementation.

Applying the knowledge of character recognition we can also predict the exact paragraph, labels and titles of each sections. Responsiveness of User Interface is another problem which can be solved and remains the further scope of our work. In future work, we will try to explore the possibility of extending our work by automating the whole process of generating the program code from the GUI screenshot, albeit its desired platform.

References

[1] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. arXiv preprint arXiv:1611.01989, 2016.

[2] W. Ling, E. Grefenstette, K. M. Hermann, T. Ko cisk y, A. Senior, F. Wang, and P. Blunsom. Latent predictor networks for code generation. arXiv preprint arXiv:1603.06744, 2016.

[3] Tony Beltramelli, pix2code: Generating Code from a Graphical User Interface Screenshot . arXiv:1705.07962v2 [cs.LG] 19 Sep 2017.

[4] K.I. Kim, K. Jung, H. Kim, Texture-based approach for text detection in images using support vector machines

and continuously adaptive mean shift algorithm, IEEE Transactions on PAMI 25 1631 1639, 2003.

[5] X. Chen, A. Yuille, "Detecting and Reading Text in Natural Scenes", Computer Vision and Pattern Recognition (CVPR), pp. 366-373, 2004.

[6] Q. Ye, Q. Huang, W. Gao, D. Zhao,"Fast and robust text detection in images and video frames", Image and Vision Computing 23 565576, 2005.

[7] Chen, H., Tsai, S., Schroth, G., Chen, D., Grzeszczuk, R., Girod, B. Robust text detection in natural images with edge-enhanced Maximally Stable Extremal Regions. 2011 18Th IEEE International Conference On Image Processing.

[8] J. Canny, "A computational approach to edge detection", IEEE Trans. Pattern Anal. Machine Intell., vol. PAMI-8, pp. 679-698, 1986.

[9] J. Gllavata, R. Ewerth, B. Freisleben, "Text Detection in Images Based on Unsupervised Classification of High-Frequency Wavelet Coefficients", 17th International Conference on Pattern Recognition (ICPR'04) - Volume 1, pp. 425-428, August 2004.

[10] Y. Liu, S. Goto, T. Ikenaga, "A Contour-Based Robust Algorithm for Text Detection in Color Images", IEICE TRANS. INF. SYST., VOL.E89D, NO.3 March 2006.

[11] B. Epshtein, E. Ofek, and Y. Wexler, "Detecting text in natural scenes with stroke width transform", in CVPR, 2010, pp. 2963 2970.

[12] G. Bradski and A. Kaehler, "Learning OpenCV: Computer Vision with the OpenCV Library" in O'Reilly Media, Inc., 2008, pp. 234-243.

[13] Guobo Xie and Wen Lu, "Image Edge Detection Based On Opencv" in International Journal of Electronics and Electrical Engineering Vol. 1, No. 2, June 2013.

[14] Google Inc., "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", arXiv:1704.04861v1 [cs.CV] 17 Apr 2017.