

AUTOMATED GENERATION OF MOBILE UI LAYOUT FILES VIA A CUSTOM GUI
ELEMENT DETECTION MODEL TRAINED WITH YOLOV5

by

Mehmet Doğan Altınbaş

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University

2023

AUTOMATED GENERATION OF MOBILE UI LAYOUT FILES VIA A CUSTOM GUI
ELEMENT DETECTION MODEL TRAINED WITH YOLOV5

APPROVED BY:

Assist. Prof. Dr. Tacha Serif

(Thesis Supervisor)

(Yeditepe University)

.....

Prof. Dr. Sezer Gören Uğurdağ

(Yeditepe University)

.....

Prof. Dr. Bülent Tavlı

(TOBB University of Economics and Technology)

.....

DATE OF APPROVAL: / / 20....

I hereby declare that this thesis is my own work and that all information in this thesis has been obtained and presented in accordance with academic rules and ethical conduct. I have fully cited and referenced all material and results as required by these rules and conduct, and this thesis study does not contain any plagiarism. If any material used in the thesis requires copyright, the necessary permissions have been obtained. No material from this thesis has been used for the award of another degree.

I accept all kinds of legal liability that may arise in cases contrary to these situations.

Name, Last Name

Mehmet Doğan Altınbaş

Signature

.....

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my supervisor, Tacha Serif, for their unwavering support, guidance, and encouragement throughout the course of this project. Their expertise, insight, and mentorship have been invaluable to me, and I am extremely grateful for the opportunity to work with them.

Finally, I am grateful to my family and friends for their support and encouragement throughout the process of completing this thesis. Their love and encouragement have been a constant source of strength and motivation.



ABSTRACT

AUTOMATED GENERATION OF MOBILE UI LAYOUT FILES VIA A CUSTOM GUI ELEMENT DETECTION MODEL TRAINED WITH YOLOV5

In mobile application development, building a consistent user interface (UI) might be a costly and time-consuming process. This is especially the case if an organization has a separate team for each mobile platform such as iOS and Android. In this regard, one of the most parts of the UI design task is creating a graphical user interface (GUI). Accordingly, this study aims to employ the YOLOv5 to create a custom object detection model that recognizes GUI elements in a given UI image. In order to benchmark the newly trained YOLOv5 GUI element detection model, existing work from the literature and their datasets are considered and used for comparison purposes. Accordingly, this study makes use of 450 UI samples of the VINS dataset for testing, a similar amount for validation, and the rest for model training. Then the findings of this work are compared with another study that has used the single shot detector (SSD) algorithm and VINS dataset to train, validate and test its model, which showed that the proposed algorithm outperformed SSD's mean average precision (mAP) by 15.69%. Moreover, this study proposes a framework that utilizes this custom GUI element detection model to generate iOS and Android UI layout files of a given mobile UI design. Thus, this framework uses machine learning algorithms to explore and extract the attributes of the detected GUI elements. Then, the framework engages in these attributes to generate UI layout files for iOS and Android platforms. To evaluate this framework, 5 test images from the VINS dataset were randomly selected and given as input for the framework. The framework generated iOS and Android UI layout files for each test image. Lastly, the preview of the UI layout files has been captured as screenshots and compared with its input test image using image similarity metrics. The test results revealed that the proposed framework has obtained an average structural similarity (SSIM) score of 0.731 and 0.703 on generated iOS and Android UI designs respectively. Moreover, on average, the framework has yielded UI designs with pixel-based similarity of 83.41% for iOS and 81.29% for Android platforms on given test images.

ÖZET

YOLOV5 İLE EĞİTİLMİŞ ÖZEL GUI ÖĞESİ ALGILAMA MODELİ ARACILIĞIYLA MOBİL UI TASARIM DOSYALARININ OTOMATİK OLARAK OLUŞTURULMASI

Mobil uygulama geliştirmede, tutarlı bir kullanıcı arabirimi (UI) oluşturmak maliyetli ve zaman alan bir süreç olabilir. Bu, özellikle bir kuruluşun iOS ve Android gibi her bir mobil platform için ayrı bir ekibi olması durumunda geçerlidir. Bu bağlamda, UI tasarım görevlerinin meşakkatli kısımlarından biri, bir grafik kullanıcı arayüzü (GUI) oluşturmaktır. Buna göre, bu çalışma YOLOv5'i belirli bir UI görüntüsündeki GUI öğelerini tanıyan özel bir nesne algılama modeli oluşturmak için kullanmayı amaçlamaktadır. Yeni eğitilen YOLOv5 GUI öğe algılama modelini karşılaştırmak için, literatürdeki mevcut çalışmalar ve bunların veri kümeleri dikkate alınır ve karşılaştırma amacıyla kullanılır. Bu nedenle, bu çalışma test için VINS veri setinin 450 UI örneğini, doğrulama için benzer bir miktarı ve model eğitimi için geri kalanını kullanır. Daha sonra bu çalışmanın bulguları, modelini eğitmek, doğrulamak ve test etmek için tek atış dedektörü (SSD) algoritmasını ve VINS veri setini kullanan başka bir çalışma ile karşılaştırıldı ve önerilen algoritmanın SSD'nin ortalama ortalama kesinliğini (mAP) %15,69 oranında geride bıraktığını gösterdi. Ayrıca, bu çalışma belirli bir mobil kullanıcı arayüzü tasarımının iOS ve Android kullanıcı arayüzü düzen dosyalarını oluşturmak için bu özel GUI öğe algılama modelini kullanan bir çerçeve önermektedir. Bu şekilde, bu çerçeve, algılanan GUI öğelerinin özniteliklerini keşfetmek ve çıkarmak için makine öğrenimi algoritmalarını kullanır. Ardından, çerçeve, iOS ve Android platformları için kullanıcı arabirimi düzen dosyaları oluşturmak üzere bu öznitelikleri kullanır. Bu çerçeveyi değerlendirmek için, VINS veri setinden 5 test görseli rastgele seçildi ve çerçeveye girdi olarak verildi. Çerçeve, her test görüntüsü için iOS ve Android UI düzen dosyaları oluşturdu. Daha sonra, kullanıcı arabirimi düzen dosyalarının önizlemesi ekran görüntüleri olarak yakalanmış ve görüntü benzerliği metrikleri kullanılarak giriş testi görüntüsüyle karşılaştırılmıştır. Test sonuçları, önerilen çerçevenin oluşturulan iOS ve Android UI tasarımlarında sırasıyla 0.731 ve 0.703 ortalama yapısal benzerlik (SSIM) puanı elde ettiğini ortaya koydu. Ayrıca çerçeve, verilen test görüntülerinde ortalama olarak iOS için %83.41 ve Android platformları için %81.29 piksel tabanlı benzerliğe sahip UI tasarımları sağlamıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
ABSTRACT	v
ÖZET	vi
TABLE OF CONTENTS	vii
LIST OF FIGURES	ix
LIST OF TABLES	xi
LIST OF SYMBOLS/ABBREVIATIONS	xii
1. INTRODUCTION	1
2. BACKGROUND	3
2.1. GUI ELEMENT DETECTION	3
2.2. GUI CODE & UI LAYOUT FILE GENERATION	7
3. METHODOLOGY	13
3.1. DEVELOPMENT TOOLS	13
3.2. ALGORITHMS	15
3.3. DATA ANNOTATION TOOLS	18
3.4. DATASETS	20
3.5. OBJECT DETECTION EVALUATION METRICS	22
3.6. UI LAYOUT FILES	24
3.7. IMAGE SIMILARITY METRICS	27
4. GUI ELEMENT DETECTION MODEL	30
4.1. ANALYSIS & DESIGN	30
4.2. IMPLEMENTATION	31
4.2.1. Preparing the Dataset	31
4.2.2. GUI Element Detection Model	32
4.3. RESULTS & FINDINGS	34
5. UI LAYOUT FILE GENERATION	38
5.1. ANALYSIS & DESIGN	38
5.2. IMPLEMENTATION	41
5.2.1. Extracting GUI components	42
5.2.2. Detecting Text and Color Attributes of GUI Elements	43

5.2.3. Generation of the UI Layout Files	44
5.3. RESULTS & FINDINGS	47
5.3.1. Structural Similarity Index Measure (SSIM)	48
5.3.2. Pixel-based Similarity	49
6. CONCLUSION & FUTURE WORK	51
REFERENCES	54
APPENDIX A	61
APPENDIX B	62
APPENDIX C	63
APPENDIX D	64

LIST OF FIGURES

Figure 2.1.	Overview of Ge's method	5
Figure 2.2.	Overall flow of Zhu et al.'s proposed system	6
Figure 2.3.	Training flow of pix2code	7
Figure 2.4.	Proposed system for automatic HTML code generation	8
Figure 2.5.	Overview of REMAUI	9
Figure 2.6.	Automatic GUI code generation scheme on sketch image	10
Figure 2.7.	An example of GUI skeleton generation from UI image	12
Figure 2.8.	The results of pix2code and PixCoder models for a given Android image	12
Figure 3.1.	Fast R-CNN architecture	17
Figure 3.2.	LabelImg annotation process	18
Figure 3.3.	Example of image annotation process in VIA	20
Figure 3.4.	Intersection over union formula	23
Figure 3.5.	Confusion matrix for machine learning	23
Figure 3.6.	Example of storyboard transitions in iOS	25
Figure 3.7.	Example of Android XML file	26
Figure 4.1.	An example of conversion from Pascal VOC XML to YOLO TXT	32
Figure 4.2.	Distribution of class labels in the training dataset	33
Figure 4.3.	A snapshot from the training phase	34
Figure 4.4.	Running inference on an image using the trained model	35
Figure 4.5.	Accuracy results of model training on the validation set	35

Figure 4.6.	Confusion matrix based on 450 test images	37
Figure 5.1.	Sequence diagram of the proposed system	39
Figure 5.2.	Activity diagram of the proposed system	40
Figure 5.3.	Overview of the UI layout file generation system	41
Figure 5.4.	The annotation file for the input screenshot	42
Figure 5.5.	Cropped GUI components from input screenshot	43
Figure 5.6.	Steps of extracting text in GUI components	44
Figure 5.7.	Detecting dominant colors in the GUI element	45
Figure 5.8.	iOS code snippet generation	45
Figure 5.9.	Android code snippet generation	47
Figure 5.10.	Selected test images	47
Figure 5.11.	SSIM calculation script	48
Figure 5.12.	Pixel-based comparison of the first test image and its generated iOS design....	50
Figure D.1.	Generation of Android UI layout file for the image in Figure 5.10a.....	64
Figure D.2.	Generation of iOS UI layout file for the image in Figure 5.10a.....	65

LIST OF TABLES

Table 4.1. Average precision (AP) at IoU of 0.5 for each of the 12 class labels on the test set.....	36
Table 5.1. Android and iOS equivalents of GUI components detected by the GUI element detection model	46
Table 5.2. SSIM results between test images and its generated iOS & Android UI designs	49
Table 5.3. Pixel-based similarity between test images and its generated iOS & Android UI designs	50
Table A.1. Designated class numbers for each class label	61

LIST OF SYMBOLS/ABBREVIATIONS

AP	Average Precision
ANN	Artificial Neural Network
ATG	Activity Transition Graph
CLI	Command Line Interface
CNN	Convolutional Neural Network
CV	Computer Vision
DL	Deep Learning
DNN	Deep Neural Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Units
GUI	Graphical User Interface
HCI	Human Computer Interaction
HOG	Histogram Oriented Gradients
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IoU	Intersection over Union
LSTM	Long Short-Term Memory
mAP	Mean Average Precision
MSE	Mean Squared Error
NCC	Normalized Cross-Correlation
OCR	Optical Character Recognition
OpenCV	Open-Source Computer Vision Library
RPN	Region Proposal Network
RNN	Recurrent Neural Network
R-CNN	Region-based Convolutional Neural Network
SPP-YOLO	Spatial Pyramid Pooling YOLO
SSD	Single Shot MultiBox Detector
SSIM	Structural Similarity
UI	User Interface

XML	Extensible Markup Language
YOLO	You Only Look Once



1. INTRODUCTION

Nowadays smartphones are much more capable than the computer that guided the first man to the moon [1]. Indeed, smartphones these days have powerful processors, advanced cameras, precise sensors, and user-friendly operating systems. Currently, there are more than 6 million mobile apps available in leading app markets for smartphones, tablets, and other devices running on different operating systems [2]. The revenue generated by these mobile apps is expected to be more than \$613 billion by 2025. That is almost double the revenue generated in 2020 [3].

One of the most cumbersome parts of mobile application development is the design of its UI. There are multiple steps that need to be undertaken to build and design a user-centered and user-friendly interface that would be easy to use by its target audience. Hence, depending on the target group - e.g. young children, teenagers, professionals, elderly, etc. - of the user base, the UI is created with different GUI element sizes, color schemes, and haptics. Developing a GUI is one of those repetitive tasks since every mobile platform has its own UI design editor, programming or markup language. There are several techniques - such as native development and cross-platform development - to implement mobile applications and their UI. Cross-platform applications are the ones that can run on multiple platforms with little or no modification. On the other hand, native apps are built specifically for a particular platform or operating system. Essentially, when these kinds of apps are built, they are designed to run on one specific platform. So, if an app is made for Android, it will not work on iOS or vice versa. Therefore, work done for one platform is required to be duplicated for another. There are a considerable number of studies in the literature that have proposed tools and frameworks to improve GUI development processes by addressing GUI element detection and automated UI layout file generation. These studies utilize current or newly emerged algorithms or models to achieve this automation.

Bearing in mind that, GUI element detection in mobile UI images is a domain-specific object detection task, there is still room for improvement based on the cutting-edge algorithms in the area of GUI object detection. To the best of our knowledge, no work has utilized the YOLOv5 object detection algorithm for mobile GUI element detection. In addition to that, there are rarely analyzed in the literature as the authors propose both a new GUI detection model and

a UI layout file generation system within the same research. Thereby, the focus of this study is two-fold; The first objective is to create and train a model that identifies a set of known GUI elements and detects their location within a UI image. Secondly, this study proposes a method that carries out UI layout file generation for both Android and iOS platforms with the help of the GUI element detection model achieved in the first part.

Accordingly, this thesis is structured as follows; Chapter 2 provides a brief introduction to GUI element detection and UI layout file generation. Chapter 3 details the tools, algorithms, datasets, and performance evaluation metrics. Chapter 4 elaborates the proposed GUI element detection model and Chapter 5 focuses on the UI layout file generation system. Finally, Chapter 6 draws the overall conclusions and discusses future development areas.

2. BACKGROUND

There is a considerable body of literature on GUI element detection and UI layout file generation. This chapter presents a review of recent research to understand more about these two domains. In fact, a large of existing studies were conducted to address issues, gaps, and shortcomings. Correspondingly, several methods have been assessed to overcome challenges in GUI element detection and UI layout file generation. These methods mostly seek to take the advantage of computer vision algorithms, machine learning, and deep learning approaches.

2.1. GUI ELEMENT DETECTION

By making use of 50000 GUI images, Chen et al. [4] attempted to undertake an empirical comparative study using the seven most popular GUI detection methods to evaluate their capabilities, limitations, and effectiveness. The findings of this study point out that existing computer vision (CV) techniques are inadequate in extracting the unique characteristics of GUI elements and detecting their location with high accuracy. So, the authors propose and implement a hybrid prototype, where they combine both the CV methods and deep learning models to test the hybrid method's extraction and location accuracy. The proposed prototype uses CV methods to detect non-text GUI elements and deep learning models for text-based GUI elements. As a result, their findings indicated that the proposed system performed 86% better in the accuracy of region classification for non-textual GUI elements; whereas achieved 91% in region classification of all elements.

Similarly, Xie et al. [5] suggested a tool that utilizes CV and object detection methods to provide accurate positioning of GUI elements along with their types. This tool is designed to assist GUI reverse engineering or GUI testing operations. Accordingly, in their work, they take the advantage of CV algorithms in detecting the position and framing the non-text GUI elements. Algorithms such as flood-filling [6], Sklansky's [7], and shape recognition are used to obtain layout blocks and as a follow-up; the connected component labeling technique is applied to identify the region of the GUI components. Following that, the ResNet50 classifier, trained with 90000 GUI elements, is then used to recognize and group each GUI element in predefined regions into 15 categories. On the other hand, in order to detect and

expose text-based GUI elements, they utilize the EAST text detector. Consequently, it has been determined that the suggested tool produces more accurate identification results than the similar studies in the literature.

Bunian et al. [8] proposed a visual search framework that takes a UI design as an input and returns design samples that resemble visually. A part of this study includes creating a custom GUI element detection model. This model is generated by using a custom dataset of 4543 UI images and adopting the SSD (Single Shot MultiBox Detector) high-accuracy object detection algorithm. It is used to find types of GUI components and their positions in the input image. The GUI element detection model is tested using 450 images from their custom dataset and it has achieved a mean average precision (mAP) of 76.39%.

Chen et al. [9] proposed a GUI component library that aids designers to discover practical use of certain GUI designs in real applications. This library consists of GUI components that were retrieved from half a million GUI screenshots of two different datasets. In the first dataset, there are 68,702 screenshots of GUIs of 5,043 Android apps. Accordingly, a method for automating UI exploration for Android apps has been developed based on the GUI testing framework [10]. It is used to identify the GUI components and their type, size, and position in the screenshots. In other words, the GUI screenshots of the first dataset were automatically annotated. In addition, 469,177 GUI screenshots of 126,298 Android apps were crawled from Google Play to create a second dataset. Based on the first dataset, a Faster R-CNN model is trained to detect GUI components, and the trained model is then used to decompose the app screenshots of the second dataset into GUI components. The evaluations showed that the trained model achieved 76% for precision, 62% for recall and 51% for mAP.

Last but not least, Chen et al. [11] proposed a framework that takes a UI image as input and generates GUI code for the requested target platform. This study aimed at lowering the development and maintenance cost of GUI code generation for both Android and iOS platforms. The implemented prototype is composed of three processes. The first process entails the component detection task, which is performed using CV algorithms such as Canny edge [12] and Edge-dilation [13]. Additionally, a convolutional neural network (CNN) is used to train two classification models based on iOS and Android platforms to identify the types of the detected GUI components. The second process involves the type mapping of the detected

GUI components for iOS and Android platforms. And lastly, the final process encompasses the GUI file generation, which is accomplished by using individual GUI code templates for Android and iOS apps. The test results of the prototype system showed that it achieved 85% accuracy in GUI element classification and managed to create UI designs that are 60%-70% compatible and similar to the original UI image.

On the other hand, Ge's approach [14] aims to find visually similar high-fidelity Android UI screens for a given UI Sketch. The goal of the proposed method is to help designers and developers to gain enthusiasm by making several corrections and alterations in their designs at the beginning stage of mobile app development. The proposed method (Figure 2.1) consists of three steps: (I) A sketch parser processes and transforms hand-drawn sketches into intermediate form; (II) A DL model obtains GUI skeletons of sketches; (III) Then, similar GUIs are identified by computing a similarity score between the generated GUI skeletons and the ones in the app repositories. A preliminary experiment was conducted on 60 Android apps from different categories. The apps cover at least one or more of the six common UI cases: login and registration, search, setting, list, picture, and detail. For each UI scenario, an app was selected and drew sketches for the relative GUIs. Six hand-drawn sketches were processed, and the results showed that the method can find the target GUIs in the top six candidates, with three ranked highest.

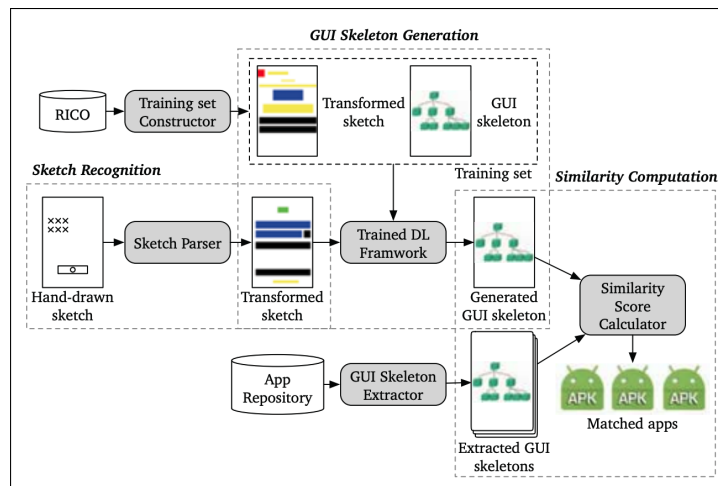


Figure 2.1. Overview of Ge's method [14]

Cheng et al. [15] employed the YOLOv5-MGC algorithm to improve the recognition accuracy of the GUI elements through the target anchor box and attention mechanism. In fact,

this study proposed to overcome some of the limitations in the existing object detection algorithms such as low accuracy rate and neglected small GUI elements. To avoid the limitations, the YOLOv5-MGC algorithm was developed as a robot vision-based interface element recognition technique for mobile applications. The algorithm enhances the network by applying the attention mechanism, adding a microscale detection layer, employing the K-means++ algorithm [16] for target anchor box generation, and adding the Ghost bottleneck module. In conclusion, the experimental results reveal that the YOLOv5-MGC algorithm outperforms the YOLOv5 algorithm for object detection and GUI element recognition. Also, the mean AP value at 0.5 IoU threshold was calculated at 89.8% and the recognition precision was evaluated at 80.8%.

Zhu et al. [17] pointed out that aerospace control software testing is a crucial process whose unforeseen or unminded scenarios may lead to insufficient product quality, jeopardizing someone's safety, and even life-threatening conditions. Also, the authors claimed that automation software testing strategies are increasingly demanding for aerospace control software and whereas traditional testing methodologies have been struggling to fulfill development requirements. Having said that, Zhu et al. also expressed that one of the most fundamental issues in the testing of the aerospace control software is to locate GUI components and their intent from the software screenshots.

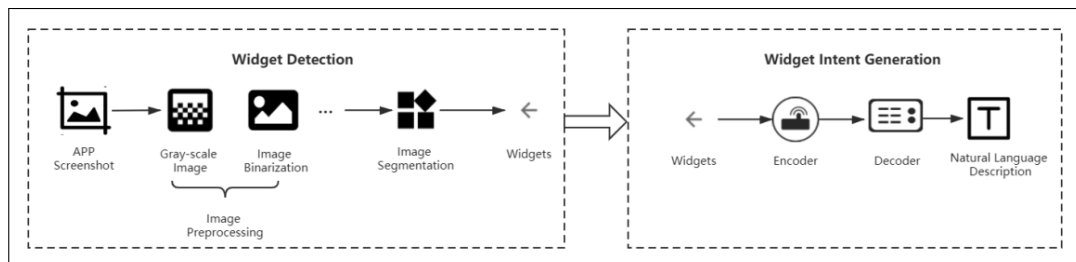


Figure 2.2. Overall flow of Zhu et al.'s proposed system [17]

Accordingly, a method was devised for detecting widget areas based on image understanding and analysis. The process starts with preprocessing the image and segmenting it to identify the layout blocks. This was followed by marking their positions in the image, calculating their hierarchical relationships, and checking whether the blocks contain widgets. The widgets' intent is generated after the screenshot was extracted from the screenshot. In order to determine widget intent, an encoder-decoder model was used. In order to reduce the amount

of vocabulary that needs to be processed, a pre-trained VGG16 model [18] is applied to recognize image features. Then the encoder module was applied to encode this extracted information feature into a tensor. According to the encoded information, the decoder module produced one word and fed the previously generated word sequence as input. As a result, the accuracy of widget recognition and classification was evaluated. This study obtained 81% of IoU for widget recognition and also effective results on intent generation.

2.2. GUI CODE & UI LAYOUT FILE GENERATION

Beltramelli [19] proposed a model — called *pix2code* — that draws advantage from reverse engineering to produce user interface code from a single GUI image input. This study aimed to reduce the time spent implementing client-side software based on a GUI mockup and ensure that developers have enough time to implement the functionality and logic of the software they build. The training phase Figure 2.3 consists of 3 neural network models. In-depth, a CNN takes input GUI mockup and generates a vectorial representation of it. Meanwhile, Long Short Term Memory (LSTM) is used to learn the computer language. Lastly, a secondary LSTM takes the input of the previous 2 models as concatenated vectors. It is claimed that the model generates code from a single image of different platforms — such as iOS, Android, and web-based technologies — with over 77% of accuracy.

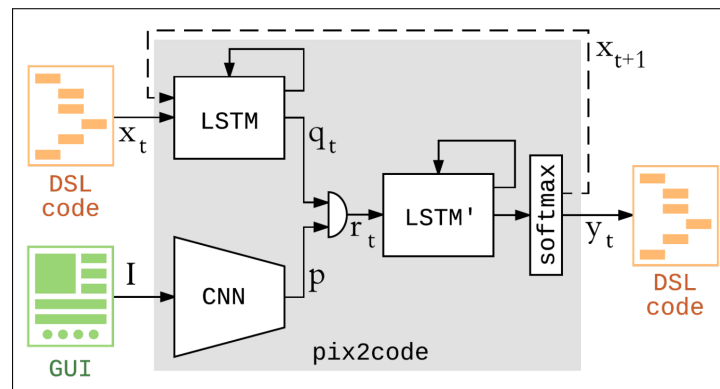


Figure 2.3. Training flow of *pix2code* [19]

Aşıroğlu et al. [20] proposed a system (Figure 2.4) to automate the HTML code generation process from hand-drawn mockups using CV methods and deep learning algorithms. The main goal of this study was to reduce programming time, and resource consumption and

accordingly finalize HTML website implementation faster. To accomplish this, image processing techniques — such as dilation [13] and contour detection [21] — were applied to identify objects in a given sketch. Also, a CNN model was trained using a custom dataset that only contains images of four types of components was used to label the identified objects. Then, the identified objects were cropped and transferred to the CNN model. After the CNN model recognized the components, they are converted to HTML code via the bootstrap framework. The model was tested after the training phase of 200 epochs and it is declared that the system achieves 96% method accuracy and 73% validation accuracy.

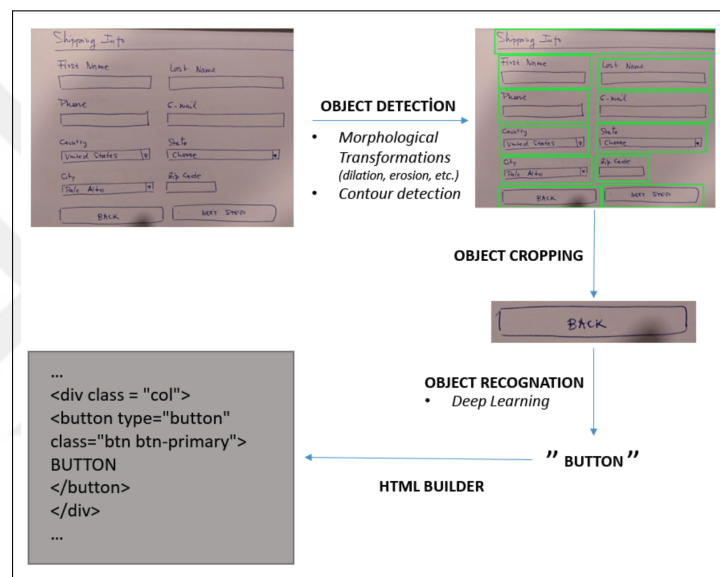


Figure 2.4. Proposed system for automatic HTML code generation [20]

On the other hand, Nguyen et al. [22] proposed a system (Figure 2.5) that automatically identifies UI components from iOS or Android app screenshots and generates user interfaces that closely resemble the original screenshots. The authors of this study intend to automate the UI code generation based on existing app screenshots. Accordingly, the proposed system applies CV and optical character recognition (OCR) algorithms on a given screenshot image. After that, overlapping areas and text blocks are rearranged by adhering to a set of rules that were defined by the authors. This rearrangement step is followed by UI element classification and app generation. Their evaluations showed that the proposed system produces very similar UIs to the original input image. Also, despite its high precision, the prototype system was able to generate UI code for each given screenshot in 9 seconds on average, which is a praiseworthy performance according to the literature.

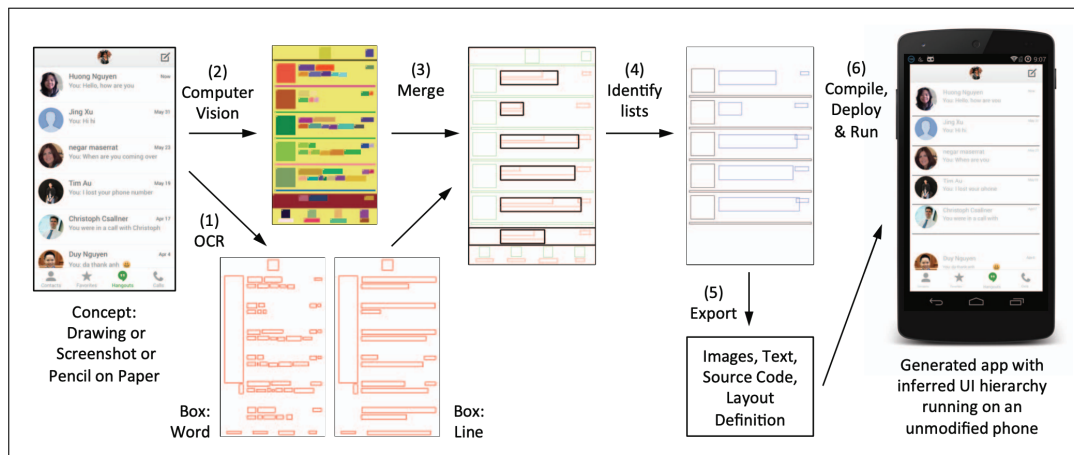


Figure 2.5. Overview of REMAUI [22]

Moran et al. [23] automatically created GUIs for mobile apps using a DL-based methodology. This study put forth the REDRAW algorithm, which makes it easier to create mobile app source code from UI design mockups. This method uses a deep CNN to identify GUI elements, which may then be utilized to produce a mock GUI for a particular app. This approach took place in 3 steps. In the first step, computer vision techniques to identify specific GUI components. The second step entails classifying the discovered components based on their purposes, such as Spinner, Button, and Switch. Deep convolutional neural networks are utilized at this step. The XML code is produced in the last step by merging the KNN algorithm considering the web programming hierarchy. The evaluation of the proposed solution revealed that REDRAW delivers an average GUI-component categorization accuracy of 91% and assembles prototype programs that closely resemble target mock-ups in terms of aesthetic affinity while demonstrating adequate code structure.

Chen et al. [24] conducted a study that suggests a system called StoryDroid, achieves generating storyboards for Android apps and assists various roles in efficiently reviewing apps. It enables app development teams, including project managers, designers, and developers, to quickly gain an understanding of other similar apps. StoryDroid generates app storyboards automatically in three stages: (I) extracting relatively entire activity transition graph (ATG), (II) statically rendering UI pages, (III) and inferring activity names for obfuscated apps. The findings showed that, on both open-source and closed-source apps, StoryDroid extracts approximately twice as many activity transitions as the leading ATG extraction tool. Also, in the tests conducted, Storydroid achieved 87% and 74% activity coverage for open-source

and closed-source applications, respectively. Last but not least, the experiments showed that StoryDroid can generate semantic names with 92% of accuracy.

Yun et al. [25] used the YOLO [26] object detection algorithm to recognize elements in a mockup by employing a proprietary annotation language that increases element type recognition. The authors' ultimate goal is to create automatic or semi-automatic code conversion and structuring of GUI components. Therefore, in this paper, a system was proposed to detect GUI elements using deep neural networks (DNNs) to represent GUI components hierarchically. To achieve this, three steps had been processed — respectively data preparation, objection detection, and code generation. To start with, LabelImg [27] is used for data annotation, and accordingly, 600 GUI elements from 50 sketch images were manually annotated along with bounding boxes. The resulting annotations are saved as an XML file in PASCAL VOC format. Secondly, the GUI element detection model is developed by making use of the YOLO object detection network. The training phase took advantage of pre-trained convolutional weights on ImageNet [28] to effectively train the model with its limited dataset. In the third step, following the detection of GUI elements, the bounding box information GUI elements were elaborated to reveal their relative position to each other. As it is depicted in Figure 2.6, this information was translated to platform-specific GUI codes.

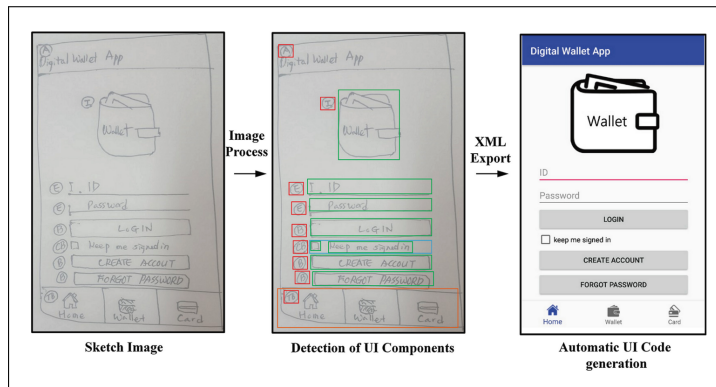


Figure 2.6. Automatic GUI code generation scheme on sketch image [25]

As a goal of many other studies in the literature, Bhambure et al. [29] proposed an end-to-end trained model that aims to reduce the time-consuming GUI development process as it is a repetitive task for an application when it is required to be run on multiple platforms. The proposed model was developed using deep learning techniques and algorithms such as CNN and Gated Recurrent Units (GRU) [30]. In addition, the training phase of the model was

carried out by using the same dataset in the pix2code [19] model. Also, this study utilized Python as the programming language, Windows 10 as the operating system, Keras [31] and Tensorflow [32] to build a neural network. However, the authors did not share any quantitative results about the overall performance of the proposed model. Instead, the study pointed out that the performance comparison would be future work.

Feng et al. [33] adopted an approach that helps developers to reduce the development time of icon design in the UI. The study consists of two parts: (I) a heuristic machine learning technique for icon composition, and (II) a deep learning and computer vision method for icon implementation. In the first part, the Hierarchical Agglomerative Clustering (HAC) algorithm [34] was used to compose the components from design artifacts into icon images. On the other hand, in the second part, a deep learning model and computer vision methods were proposed to convert composed icons into fonts along with descriptive labels, thus simplifying UI design and reducing developer workload. As a result, this study quantifies the proposed method's performance in the real-world UI development environment by demonstrating that it saves 65.2% of implementing time in UI development compared to other methods.

Chen et al. [35] presented an automated GUI exploration technique for addressing the first challenge in the lack of variety of UI designs and GUI skeletons. Then, a neural machine translator was proposed to handle the second challenge of visual understanding and skeleton generation through the combination of recent advances in computer vision and machine translation. This study aims to reduce the complexity of GUI implementation. A large dataset of diverse screenshots and GUI skeletons is automatically collected and used to train the neural machine translator. Accordingly, 185,277 pairs of UI images and GUI skeletons were automatically collected from 5043 Android apps using an Android UI data collector. The dataset is used to train the neural machine translator and to evaluate its accuracy in generating UI skeletons from UI images. The evaluations showed that the suggested translator accurately produces GUI components and compositions for a wide range of GUI skeleton complexity by distinguishing different types of visual elements and spatial layouts in UI images.

Huang et al. [36] developed a system, PixCoder, to increase the accuracy of GUI code generation by the help of deep learning techniques. Also, this study came up with a solution to deal with issues in pix2code [19] due to the lack of attention mechanism. PixCoder is

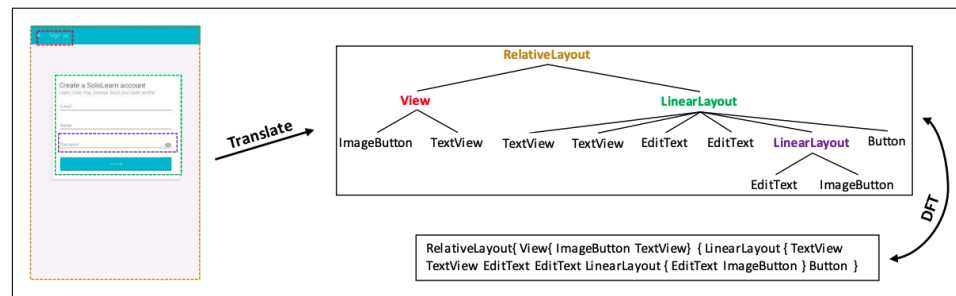


Figure 2.7. An example of GUI skeleton generation from UI image [35]

a tool that is based on a method that utilizes artificially supervised attention mechanisms to solve the problems in pix2code. To further understand, this approach includes a trained neural network to recognize the style sheet in the input GUI image to create an output vector. PixCoder produces GUI code for the particular platform regarding the output vector. The experiments revealed that the PixCoder outperforms pix2code in GUI code generation. As it can be seen in the Figure 2.8, the GUI code generation performance of pix2code and PixCoder were compared from a given UI image (Figure 2.8a), and, the errors were denoted in red frames. As a result pix2code (Figure 2.8b) generated GUI code with falsely inferred eight GUI components. On the contrary, PixCoder (Figure 2.8c) has only one error in the result.



Figure 2.8. The results of pix2code and PixCoder models for a given Android image [36]

3. METHODOLOGY

In line with the literature detailed above, this chapter discusses the tools, algorithms, datasets, and evaluation metrics that are used for GUI element detection.

3.1. DEVELOPMENT TOOLS

There are numerous approaches to implementing a GUI element detection model. Accordingly, there are many development environments and an extensive set of libraries that are built upon various programming languages. For example, OpenCV (Open-Source Computer Vision Library) [37] is a free and open-source software library for computer vision and machine learning. OpenCV provides a common infrastructure for computer vision applications. It supports several programming languages and platforms. Additionally, Apple has recently launched its CV framework called Vision [38], which enables developers to utilize computer vision algorithms. Also, Google Colab [39] has been developed as an open-source project to provide researchers with a fully functional computing environment, where they can write and execute arbitrary Python code for machine learning tasks without needing to set up a whole computing infrastructure.

Tensorflow [32] is an open-source software library for machine learning that was developed by Google. It allows developers to create custom algorithms to optimize and improve their machine-learning models. This library works by providing developers with a set of tools that they can use to create and train machine learning models. These tools include a programming language (Python), a library of pre-built algorithms (Tensorflow), and a set of hardware devices (GPUs). There are two ways to use Tensorflow: via the web interface or through the command line interface (CLI). The web interface is best suited for beginners, as it provides a graphical user interface that can be used to create and train machine learning models. The command line interface is best suited for advanced users, as it allows developers to directly access the Tensorflow library and create custom algorithms.

In addition, PyTorch [40] is a powerful open-source tool for deep learning that can be used to implement a wide variety of object detection algorithms. It offers many advantages over other

deep learning frameworks, including its flexibility and ease of use. Furthermore, PyTorch has been used to implement several state-of-the-art object detection algorithms, including Faster R-CNN and YOLO. This tool is an active open-source project with strong community support, making it a good choice for those looking to develop object detection applications.

Caffe [41] is a deep learning framework that is widely used for image classification and object detection tasks. The Caffe framework makes it easy to train and deploy deep learning models on a variety of devices, including CPUs, GPUs, and FPGAs. Also, the Caffe framework includes a number of pre-trained deep learning models that can be used for object detection tasks. Before the Caffe framework is used, it must be installed on Linux, macOS, or Windows systems.

Moreover, IBM Watson Visual Recognition tool [42] is another solution to achieve object detection tasks. Watson Visual Recognition can be used to detect objects in images and return information about those objects, such as the object's name, bounding box, and confidence score. Additionally, Watson Visual Recognition can be used to perform image classification tasks such as identifying landmarks or classifying images into categories.

Google Cloud Vision [43] is a cloud-based image recognition service that provides developers with access to powerful image analysis tools. The service offers a number of features, including object detection, facial recognition, and landmark identification. Google Cloud Vision is based on the same technology that powers Google Image Search and is able to identify over 10,000 different objects. Google Cloud Vision works by analyzing images and extracting information from them. The service uses a variety of machine learning algorithms to identify features in images, such as faces, objects, and landmarks. Once an image has been analyzed, the results are returned to the developer in the form of JSON data.

Keras [31] is a high-level API that allows users to easily build and train deep learning models. Keras is written in Python and utilizes the TensorFlow infrastructure. With Keras, users can create both sequential and functional models. Sequential models are easy to understand and can be created by stacking layers sequentially. Functional models are more flexible and allow for creating custom layers and models. Keras also comes with a number of pre-built layers that can be used to create models quickly. In addition, Keras includes a number of utilities that make working with deep learning models easier, such as a data generator and a

visualization tool. Overall, Keras makes working with deep learning models much simpler and faster, allowing users to focus on tasks such as model development and training.

Last but not least, Amazon Rekognition [44] is a machine learning service that makes it easy to add image and video analysis to your applications. With Rekognition, researchers can identify objects, people, text, scenes, and activities in images and videos, as well as track the locations of objects in videos. Amazon Rekognition also provides highly accurate facial recognition and facial analysis features. For example, one can use facial recognition to verify the identity of a person or to find similar faces in a large image database. You can also use facial analysis to detect facial features such as landmarks, pose, and emotion. Amazon Rekognition makes it easy to build machine learning-powered applications for large organizations.

3.2. ALGORITHMS

It is undoubtedly true that object detection tasks can take advantage of image processing algorithms. These algorithms can be used to filter out redundant or less relevant information and highlight the desired parts and important features of the image, which can be an invaluable asset in GUI element detection. Accordingly, an image in hand can be sharpened or its contrast can be increased using these algorithms, which as a result it would end up in a much clearer form with highlighted edges, finer details, and more explicit regions. Also, image processing algorithms can be used to flip, rotate, scale, or crop images to achieve data augmentation and increase the amount of data that can be used while an object detection model is trained. Image processing techniques can be considered as an intermediary step in the preparation and enhancement of a raw input image before it is processed with object detection algorithms to derive some meaningful information.

Even though numerous studies have combined and utilized the aforementioned algorithms successfully, machine learning techniques also produce promising results when used for GUI element detection. For instance, HOG [45], or Histogram of Oriented Gradients, is a feature descriptor used in computer vision and machine learning. HOG has been demonstrated to be effective for a variety of object detection tasks, including pedestrian detection, face detection,

and vehicle detection. To briefly explain, HOG works by computing the gradient of an image (the change in intensity values) in both the x and y directions. These gradients are then quantized into a set of binned angles. The resulting histogram is then used as a features vector for training a classifier.

LSTM [46] is a Recurrent Neural Network (RNN) [47] model that is used in various areas of natural language processing, including speech recognition and machine translation. LSTM can help to solve problems such as long-term dependency, which previous RNNs were not designed to address. LSTMs are useful for tasks that require remembering information over long periods. For example, LSTMs were applied to the task of learning word embeddings for a character-level language model in order to learn representations for sentences or documents. In object detection, we use LSTMs to extract features from images. The LSTM then trains on these extracted features (a process called feature extraction) and then uses those features to predict bounding boxes and labels in images. The outputs of the LSTM can be processed by a simple feedforward convolutional neural network which predicts bounding boxes and their confidence scores.

Additionally, deep learning, which is a subdomain of machine learning, introduces a more accurate way of computer vision as it uses a more sophisticated algorithm called artificial neural network (ANN). An ANN mimics the human brain as it consists of millions of interconnected processing nodes that correspond to similar behavior to neurons in the brain. Therefore, ANNs are broadly used to make generalizations and inferences. Also, it is one of the best techniques to find patterns, make predictions and reveal the relationships between the input and the output of any given problem.

There are several types of neural networks depending on the context of their use; However, convolutional neural networks (CNNs) [48] are the ones that are most utilized in object detection. CNNs vary based on the procedure they follow. For instance, in region-based convolutional neural networks (R-CNN) [49], an image is divided into region proposals and then CNN is applied for each region to extract the most fundamental features. Although this approach produces good results, the training phase is long and time-consuming since it applies CNN to each one of the regions one by one.

On the other hand, Fast R-CNN [50], is a method of object detection that is proposed to solve

the aforementioned drawbacks of R-CNN. The key to its speed is the use of region proposals. Rather than scanning the entire image for potential objects, Fast R-CNN only considers a small region at a time. This significantly reduces the amount of computation required and allows the algorithm to run in real-time. In addition, Fast R-CNN is able to learn complex relationships between pixels, making it more accurate than previous methods. Also, Fast R-CNN has become a state-of-the-art approach for applications that require real-time object detection, video surveillance, or self-driving cars. Besides, Faster R-CNN [51] takes an even further step to improve speed by using a region proposal network (RPN) instead of a selective search algorithm. The RPN is trained end-to-end with the rest of the network, making it faster and more accurate than Fast R-CNN.

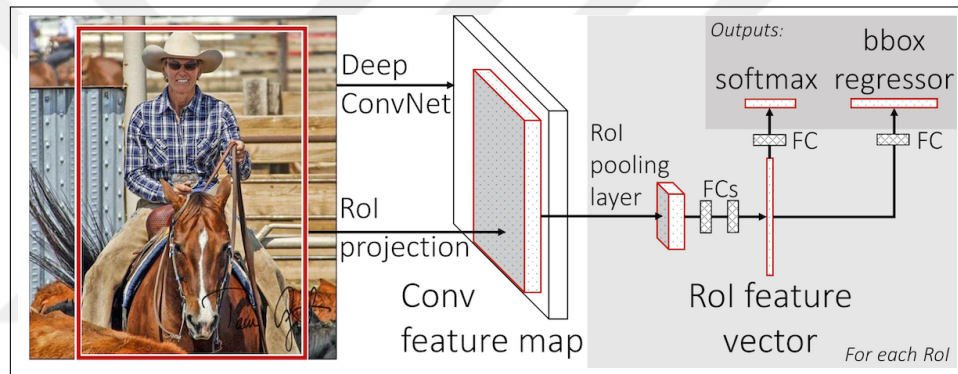


Figure 3.1. Fast R-CNN architecture [50]

Moreover, Single Shot Detector (SSD) [52] is a deep learning model that is trained on images in an unsupervised fashion. The model learns to detect objects by looking for patterns in the images. Once the model has been trained, it can detect objects in new images. SSD has shown promising results on several object detection benchmarks. In particular, it has outperformed traditional supervised methods on the COCO dataset. One advantage of SSD is that it can be trained on large amounts of data very quickly. This is due to the fact that the model only needs to be trained once, on all of the data at once. Furthermore, SSD can be used to learn from both labeled and unlabeled data. This makes it very flexible and efficient for object detection tasks.

Last but not least, You Only Look Once (YOLO) [53] is one the most popular algorithms for object detection. YOLO stands out from the above-mentioned algorithms as it seeks parts of the image rather than the complete image and predicts the bounding boxes of objects and

their class probabilities. Also, YOLO uses a single neural network to predict both bounding boxes and class probabilities for objects in an image. The algorithm is able to process images very quickly and can run predictions on multiple objects in real-time.

3.3. DATA ANNOTATION TOOLS

Data annotation is the process of labeling data to mark features that a machine learning model learns patterns to produce precise results. A model which is trained with an annotated dataset is anticipated to recognize the same features from the data that the model has never learned before. Many data annotation tools are diversified their key features, pricing strategy, and deployment model.

LabelImg [27] is a free and open-source graphical image annotation tool (Figure 3.2) to label objects using bounding boxes in images. It is developed in Python and uses Qt for the graphical interface. it can be installed locally on an operating system such as Windows, Linux or Mac. Also, LabelImg saves annotations in multiple formats such as Pascal VOC XMLs and YOLO's txt. Additionally, it supports hotkeys to speed up and facilitate the annotation process.

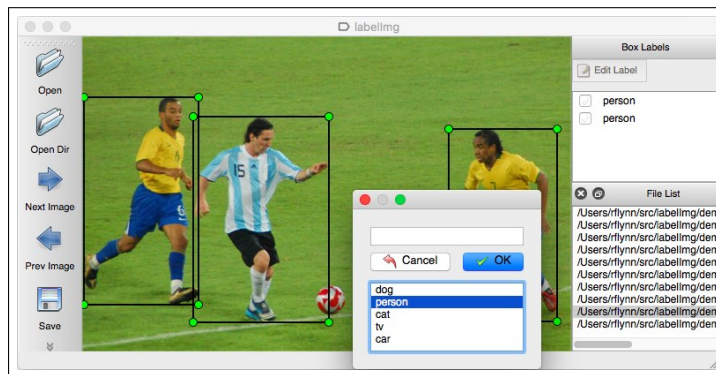


Figure 3.2. LabelImg annotation process [27]

LabelMe [54] which is developed by MIT is an online, free and open-source graphical annotation tool to build image databases for computer vision research. LabelMe is written in Python and it has Qt graphical interface. It enables various annotation types such as polygon, rectangle, circle, line, point, and image-level flag annotation. Also, it only supports LabelMe JSON format to save and export annotation files.

Computer Vision Annotation Tool (CVAT) [55] which is an OpenCV project is a web-based, collaborative and open-source annotator. CVAT supports all operating systems, meaning it can be installed locally and used offline. CVAT aids the key objectives of supervised machine learning namely, object detection, image classification, and image segmentation. This tool provides four fundamental types of annotation including boxes, polygons, polylines, and points. There is a wide selection of export formats that CVAT supports and these are most commonly used: CVAT, TFRecords, Pascal VOC, YOLO, and LabelMe.

Also, Microsoft has developed its own data annotation tool — Visual Object Tagging Tool (VoTT) [56]. VoTT provides a full suite of services to generate datasets and validate object detection models from video and image assets. Moreover, this tool is free, open-source, and available as a standalone web application and native desktop application. However, the web application requires that the dataset needs to be uploaded to a cloud as it cannot access a local file system. VoTT is limited to supporting only two annotation types: polygons and rectangles. On the contrary, it offers various export formats including CSV, JSON, Pascal VOC, and TFRecords.

Moreover, Dutta et al. [57] proposed a manual annotation software — VGG Image Annotator (VIA) (Figure 3.3) — to annotate data types such as images, video, and audio. VIA runs as an offline application in most modern web browsers. It is distributed free and available for anyone can inspect, modify, and enhance. Furthermore, It supports several annotation shapes such as bounding boxes, circles, and ellipses, as well as polygons, points, and polylines. Additionally, it provides keyboard shortcuts to accelerate the annotation process. Considering its competitors, VIA is not one of the most diverse tools in terms of the number of supported annotation formats for export. It supports output formats including COCO JSON, Pascal VOC, XML, and CSV.

On the other hand, there are several paid platforms that facilitate the annotation process. For example, Labelbox [58] is one of the most popular paid platforms that serve AI-assisted data labeling automation, data management, and Python SDK for extensibility. It annotates polygons, bounding boxes, and lines. Labelbox offers a free plan for up to 5000 images, along with the availability of pro and enterprise plans.

Additionally, SuperAnnotate [59] provides a set of tools for creating training datasets for

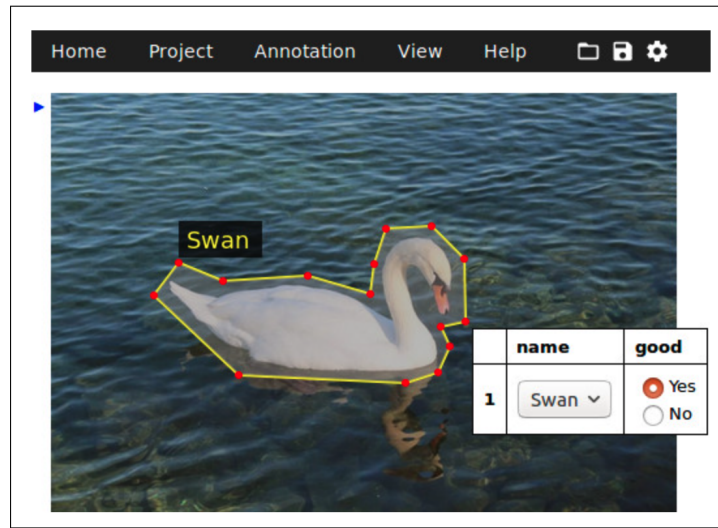


Figure 3.3. Example of image annotation process in VIA [57]

various computer vision tasks, including object detection, semantic segmentation, keypoint annotation, and video tracking. Also, it offers a free web-based tool to bring a free annotation tool to the broader computer vision community. SuperAnnotate gives free access to paid features for up to 50,000 items. In addition to that, it offers pro and enterprise plans for high-volume and sophisticated AI projects.

Last but not least, Supervisely [60] is a web-based image and video annotation platform that allows teams and individual researchers to annotate data and perform various statistical analyses on those annotations. It supports annotation to not only images and videos but also 3D point clouds built by complex sensors like LIDARs and radar sensors. Also, the platform provides a high level of extendability through the support of plugins and applications. In terms of pricing, the community edition allows 100 data labeling per day for free. Along with that, it offers business and enterprise plans.

3.4. DATASETS

There are several datasets in the literature that are used in various GUI-related studies. The Rico dataset [61] is a collection of design data that reveals the graphical, textual, structural, and interactive design attributes of Android apps. These attributes can be convenient to support data-driven applications in various ways such as design searching and UI layout

file generation. This dataset consists of 72000 individual UI screens from more than 9700 Android applications, which are created based on 27 different app categories.

Furthermore, Rico has also been a starting point for other datasets. For example Enrico (Enriched Rico) [62], which is a large dataset of mobile UIs, curated by selecting 1460 out of 10000 manually revised UIs of the Rico dataset. In the study, the Enrico dataset is used to train a deep learning model to achieve a topic classification. The results indicated that this model achieved Top-1 accuracy over 75% and an AUC of 95% when trained on UI screenshots.

Also, the pix2code dataset which is proposed and used in Beltramelli's study [19] consists of 3500 GUI screenshots along with DSL markup pairs for 3 different platforms — namely iOS, Android, and web. pix2code dataset was employed to train a model that is designed to generate user interface code from a single GUI image input.

Additionally, the VINS dataset [8] is a collection of annotated UI images and it consists of wireframe images, and screenshots from iPhone and Android apps. The UI designs in the dataset are from the Uplabs [63], and UI screens from the Rico dataset. It comprises 4800 pairings of UI design images with their matching XML files in Pascal-VOC format. The original study [8] where the VINS dataset was first introduced aimed at building an object detection-based visual search framework for a given mobile UI image. The evaluation result of the framework produced promising results in terms of GUI element detection accuracy and visual search precision.

As part of the REDRAW implementation, Moran et al. [23] collected a GUI dataset for mobile applications that include screenshots and GUI-related metadata for over 14000 screens and more than 190000 GUI components. This dataset was used to train and evaluate CNN and KNN machine learning methods in the original study.

On the other hand, there are also datasets consisting of hand-drawn sketches of various UI elements. For instance, Pandian et al. [64] proposed an open-source dataset of UI element sketches — called UISketch. This dataset is a collection of 17979 hand-drawn UI component sketches from 21 categories. 967 participants including, HCI and CS graduate students, UI/UX designers, and front-end developers, from 10 different countries took part

in the collection of these sketches. A perceptual study using this dataset found that UI/UX designers recognize UI element sketches accurately 96% of the time. Also, the recent DNN-based image classification models were trained to recognize the UI elements sketches in order to measure human performance against computational recognition methods. It is concluded that the ResNet-152 model obtained an accuracy of 91.77% in detecting unknown UI elements sketches compared to other classification networks.

3.5. OBJECT DETECTION EVALUATION METRICS

Object detection evaluation metrics reveal how well the object detection model is performing. Accordingly, precision is the ratio of correctly detected elements to all the elements detected by the model. It is a measure of how accurate a model is at predicting positive samples. As it is depicted in Equation (3.1), it is calculated by dividing the true positives by the total number of positive predictions. Also, recall is a measure of how many relevant objects are detected, which is calculated by dividing the true positive elements by the total number of true positives and false negatives - see Equation (3.2).

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \quad (3.1)$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \quad (3.2)$$

On the other hand, the intersection over union (IoU) (Figure 3.4) is the ratio that is obtained by dividing the number of pixels in the intersection between a ground-truth object and a predicted object, with the number of pixels in the union. It is mostly used as an evaluation metric for tasks such as object detection and segmentation. Also, the average precision (AP) metric is often used to assess the accuracy of object detectors such as the Faster R-CNN and YOLO. AP is calculated as the area under the precision-recall curve.

Besides, a confusion matrix Figure 3.5, also known as an error matrix, is a specific type of table that is used to evaluate the performance of a classification model. The number of correct and incorrect predictions is summed up in the table, which allows for easy analysis of the results. Each row in the confusion matrix represents the actual class, whereas each column represents the predicted class. To exemplify, if the model predicts that an image contains a

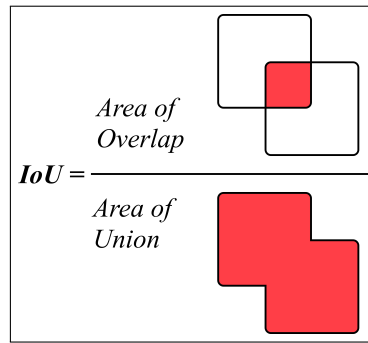


Figure 3.4. Intersection over union formula

dog, but it actually contains a cat, this would be classified as an error. The confusion matrix can be used to determine which objects are most often misclassified by the machine learning algorithm. This information can then be used to address through further training or tuning of the machine learning model.

		Predicted Values	
		Negative (N)	Positive (P)
Actual Values	Negative (N)	True Negative (TN)	False Positive (FP)
	Positive (P)	False Negative (FN)	True Positive (TP)

Figure 3.5. Confusion matrix for machine learning

However, this information can be misleading, as it does not take into account false positives and false negatives. False positives are cases where the model predicts that a given instance belongs to a certain class, when in fact it does not. False negatives are cases where the model predicts that a given instance does not belong to a certain class, when in fact it does. These two types of errors can be further divided into four types: true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). The confusion matrix can be extended to include these four types of errors, which leads to a more accurate evaluation of the model's performance. The terms "true" and "false" refer to whether or not the object is present in the image, while the terms "positive" and "negative" refer to whether or not the object was correctly detected. In other words, a true positive is an object that was both present and

correctly detected, while a false positive is an object that was present but incorrectly detected. Similarly, a true negative is an object that was both not present and correctly detected, while a false negative is an object that was not present but incorrectly detected.

Furthermore, average precision (3.3) is a metric that is often used in conjunction with precision and recall. It gives a more complete picture of a model's performance by averaging the precision (P_n) at different recall (R_n) levels. Besides, mean average precision is another metric that can be used to evaluate object detection models. It is similar to average precision, but instead of averaging the precision at different recall levels, it averages the precision at different thresholds.

$$AP = \sum_n (R_n - R_{n-1})P_n \quad (3.3)$$

Additionally, F1 score (3.4) is a metric that precision and recall into a single measure, providing a holistic view of model performance, and is, therefore, range-bound between 0 and 1. The higher the F1 score, the better the model is at detecting objects.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \quad (3.4)$$

3.6. UI LAYOUT FILES

In iOS and Android, layout files are used to define the user interface for an app. These files specify the arrangement of user interface elements such as buttons, text fields, and images, as well as their attributes such as size, position, and styling. The layout files are written in a declarative format, which means that they describe the desired appearance and behavior of the user interface, rather than specifying the exact steps that the app should take to create it.

In iOS, a storyboard is a visual representation of the user interface of an app. It is a collection of screens, or scenes, that depict the various states an app can be in and the transitions between those states. A storyboard is a useful tool for designing the flow and navigation of an app, as it allows developers to see the app's user interface and the connections between different screens. In a storyboard, each screen is represented by a scene, which is a rectangular area that contains the user interface elements for that screen. Scenes are connected by segues, which are lines that represent the transitions between screens. Segues can be triggered by

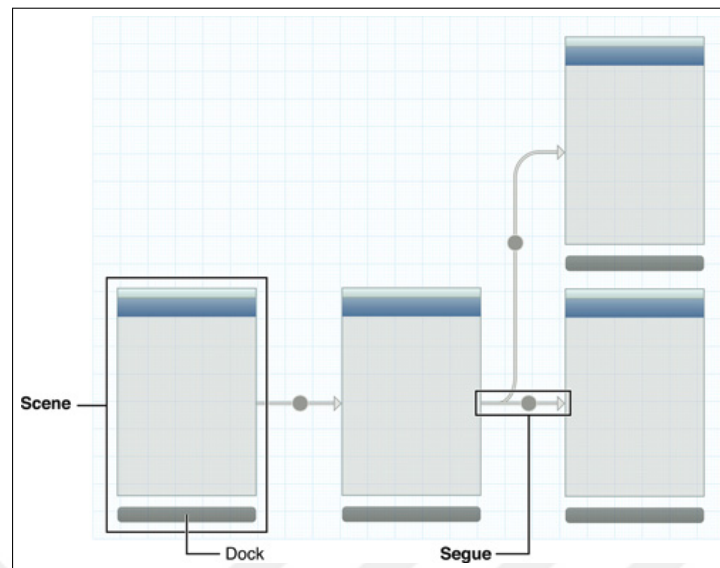


Figure 3.6. Example of storyboard transitions in iOS [65]

user actions such as tapping a button, and they can be configured to perform different types of transitions, such as pushing a new screen onto the screen stack or presenting a modal screen. Behind the scenes, a storyboard is actually a collection of XML files that define the user interface and transitions of the app. These XML files are stored in a special format called Interface Builder, which is used by Xcode to render the visual representation of the storyboard. The XML file in a storyboard defines the structure and content of each scene, as well as the connections between scenes and the transitions between them.

Some of the XML elements that a storyboard might include are `<viewController>`, `<scene>`, `<object>`, `<action>` and `<outlet>`. For example, the `<viewController>` element represents a view controller and its associated scene in the storyboard. It contains information about the view controller, such as its class and its identifier. Similarly, the `<scene>` element represents a scene in the storyboard, and it includes attributes about the scene, such as the size and position of its view. It also contains the objects that make up the user interface of the scene, such as buttons and labels.

On the other hand, the `<object>` element represents an object in the user interface of a scene. It involves the specifications of the object, including its class and its identifier. Moreover, the `<action>` element represents an action in the storyboard, which is a method that is called when a certain event occurs, such as the method that it calls and the event that triggers it.

Finally, the `<outlet>` element represents an outlet in the storyboard, which is a reference to a user interface element that can be used to manipulate that element in code.

In Android, XML files (Figure 3.7) are used to define the structure and content of user interface elements. These files are written in XML, which is a markup language for defining the structure and content of user interface elements. Android provides a number of layout managers that can be used to arrange user interface elements within an XML file. For example, a *LinearLayout* manager arranges its child elements in a horizontal or vertical line, while a *RelativeLayout* manager allows child elements to be positioned relative to each other and to the parent layout. Moreover, layout files might specify the hierarchy and attributes of user interface elements such as buttons, text fields, and images, and then reference one or more XML files to define the content of those elements. For example, an XML file might contain the text to be displayed in a button, or the image to be displayed in an image view.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Figure 3.7. Example of Android XML file

Overall, layout files are an important part of the app development process, as they allow developers to specify the visual appearance and behavior of their app's user interface in a clear and concise way. Also, layout files allow developers to separate the user interface design from the app logic, which makes it easier to modify and maintain the app over time. By keeping the user interface design in a separate layout file, developers can make changes to the user interface without having to modify the app's code. This can save time and effort, and it can help prevent errors and inconsistencies in the app's user interface.

3.7. IMAGE SIMILARITY METRICS

Image similarity metrics are algorithms used to compare images and determine their degree of similarity. These algorithms typically work by comparing various features of the images, such as their pixel values, colors, shapes, textures, and/or other visual characteristics. The goal of these algorithms is to determine how similar the images are to one another and to produce a numerical score or measure that reflects this similarity.

On the other hand, Image similarity metrics can be useful for evaluating and comparing user interface (UI) designs. For example, a designer could use an image similarity metric to compare a set of UI designs and determine which one is most similar to a reference design. This can be helpful for ensuring that the final design is consistent with the overall aesthetic and design principles of the product.

For instance, mean squared error (MSE) [66] is a metric used to measure the difference between two continuous variables. In the context of two UI designs, MSE can be used to measure the difference between the predicted values of one UI design and the actual values of the other UI design. To calculate MSE for two UI designs, the difference between each pixel value of one UI and the corresponding pixel value of the other UI is squared, and the average of these squared differences is taken. This average is then used as a measure of the overall difference between the two UIs. As it is depicted in Equation (3.5), D is the number of predicted values, x_i is the actual value for the i^{th} prediction and y_i is the predicted value for the i^{th} prediction.

$$\sum_{i=1}^D (x_i - y_i)^2 \quad (3.5)$$

Moreover, the structural similarity index measure (SSIM) [67] is a method for measuring the similarity between two images. It is often used in image and video processing to assess the quality of images and to ensure that they are visually similar to the original. The SSIM index is based on the idea that the human visual system is highly adapted to detecting changes in structural information, such as changes in spatial relationships between pixels. The SSIM index takes into account luminance, contrast, and structure, and it is designed to be more faithful to human perception than other methods for measuring image quality. To compute the

SSIM index between two images, a number of steps are performed, including the following:

- i. The two images are resized to the same size and converted to grayscale.
- ii. The mean and standard deviation of each image are calculated.
- iii. A Gaussian filter is applied to each image to remove noise and reduce high-frequency artifacts.
- iv. The two images are divided into small windows and the mean and variance of each window in both images are calculated.
- v. The SSIM index is calculated for each window by comparing the means, variances, and covariances of the two images.
- vi. The SSIM index for the entire image is calculated as the average of the SSIM indices for all of the windows.

Also, SSIM formula (3.6) combines several terms that represent the structural information of two images being compared. In the following formula:

- x and y represents the two compared images.
- μ_x and μ_y are the means of the pixel values in images x and y .
- σ_x and σ_y are the standard deviations of the pixel values in x and y images.
- σ_{xy} is the covariance between the pixel values in x and y images.
- C_1 and C_2 are constants that control the sensitivity of the SSIM metric to luminance and contrast changes.

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1) + (2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \quad (3.6)$$

The normalized cross-correlation (NCC) [68] algorithm is another method for calculating how similar two images are. It works by dividing each image into small blocks, calculating the mean and standard deviation of the pixel values in each block, and comparing these values to see how similar the blocks are. The NCC algorithm generates a score between 0 and 1, with 0 indicating that the two images are entirely different and 1 suggesting that they are identical.

NCC is frequently used in image and video processing applications because it is relatively insensitive to small shifts or translations in the images and is computationally efficient.

Blink-diff [69] is a command-line tool that can compare two images and determine whether they are visually different. It is often used in automated testing and continuous integration environments to validate that changes to a website or application have not introduced visual regressions. To use blink-diff, the paths of the two images, as well as a number of optional parameters that control how the comparison needs to be set.

One of the key features of blink-diff is its ability to handle images with different sizes and aspect ratios. It can automatically align the images and resize them as necessary to ensure that they are properly compared. This can be useful in situations where the images being compared are not exactly the same size, or where one image has been resized or cropped. Blink-diff also supports a number of image formats, including JPEG, PNG, and GIF. It can be used on a variety of platforms, including Windows, macOS, and Linux, and it is distributed as a Node.js package, making it easy to install and use.

For example, the tolerance value that how different the images are allowed to be, or a designated area of images to ignore when performing the comparison. Once the comparison is complete, blink-diff will output a report indicating whether the images are visually different, and if they are, it will highlight the differences in a third image. This image can be used to identify exactly what has changed between the two images, which can be helpful in debugging visual issues.

4. GUI ELEMENT DETECTION MODEL

This chapter details the process of building a GUI element detection model. This chapter analyzes requirements and defines a systematic approach to building a GUI element detection prototype. Also, the test cases of the prototype model are assessed and the findings are elaborated.

4.1. ANALYSIS & DESIGN

In order to create a custom GUI detection model, a collection of mobile UI images must be either newly generated or an existing dataset should be used. Furthermore, these images should be grouped in a way that they contain predominantly a specific GUI element - where each image group becomes a training class for the specific GUI element. These classes can be seen as the objects that are intended to be discovered by the object detector.

Considering that this study aims to evaluate the accuracy and the average precision of the new YOLOv5 model, there is a need for a similar study where the same classes and evaluation phases can be followed. Accordingly, the training image dataset and the algorithm that is deemed suitable for our study is the work of Bunian et al. [8] Therefore, all the image classes that need to be generated for this study, are required to follow suit the structure of Bunian et al.'s work. For the very same reason, instead of creating our own dataset of UI images, the VINS dataset which is used in Bunian et al.'s study is obtained and split into multiple subsets - i.e. training images, validation images, and testing images. In order to keep the consistency of the comparative work, this model is also to be trained, validated, and tested with the very same number of UI images of the same image dataset. In the comparative study [8], the authors make use of 4543 out of 4800 VINS UI images; where the total 4543 is split into three smaller groups as 3643 for training, 450 for validation, and 450 for testing.

Taking into consideration the above image numbers, the system that will be training the model should at least have a decent amount of CPU and GPU resources. There are multiple paths that could be taken to train this model - such as local PC-based or cloud-based training approaches. Each one of these options does contain its own advantages, such as for the

prior, all the data is stored locally and training could last as long as required. The latter does not involve running a local server or in-house PC, which can be distracted by hardware failure or power outage. However, if the training process is conducted on the cloud, then the researcher is limited with the amount of service time that is given by the cloud provider, which can be a cumbersome task if the training phases take long periods of time and require re-setup/reconfigure every time.

4.2. IMPLEMENTATION

The following chapter details the preparation and development steps taken to achieve the proposed GUI element detection model. Accordingly, to train the prototype model, a Mac mini, with 3 GHz 6-Core 8th generation Intel core i5 and 32GB memory, is used to carry out all the operational steps necessary - such as data preparation and model training. As mentioned in the previous chapter, the same training can be conducted on Google's Colab cloud solution, however, when this option was tried initially, the time required to train the model increased substantially, since the Colab virtual machine lifetime is limited to 12 hours and required reconfiguration to continue from the last checkpoint. Hence, the implementation of the prototype is proceeded on an in-house Mac mini.

First and foremost, the implementation starts with data preparation, which involved the conversion of the Pascal VOC annotation files to a YOLOv5-compatible format. Thereafter, the GUI element detection model training phase is performed.

4.2.1. Preparing the Dataset

The work by Bunian et al. [8] has two phases: (a) to identify and locate UI elements in any given screen capture image using a trained model by the SSD algorithm; (b) to use the initially created VINS image dataset to compare with the given screen capture image and identify the most similar images based on UI element hierarchies. Since this work only aims to evaluate and compare the precision of UI element identification and location using a YOLOv5-trained model, it will try to faithfully follow the steps of the first phase (a) of Bunian et al. Therefore, all class sizes and names used in the implementation will comply with the existing study.

In line with the Bunian et al. study, the total number of UI images in the dataset is reduced from 4800 to 4543. This reduction is achieved by removing the existing wireframe images in the original dataset. Also, to match the dataset structure with the existing study, 4543 UI images are split into three sets for training, validation, and testing - respectively having the ratio of 80%, 10%, and 10%. Furthermore, the naming convention of the classes that are used for training was kept in compliance with the existing study so that the results of the new model can be compared directly. Accordingly, the names of the classes generated are *Background Image*, *Checked View*, *Icon*, *Input Field*, *Image*, *Text*, *Text Button*, *Sliding Menu*, *Page Indicator*, *Pop-Up Window*, *Switch* and *Upper Task Bar*.

The VINS dataset contains annotation files along with the mobile UI images. The annotations are stored in XML files using Pascal VOC format. However, YOLOv5 demands annotations for each image in the form of a TXT file, with each line describing a bounding box. Therefore, a Python script is run in the directory where the dataset is located, to find each of the XML files (Figure 4.1a) and convert them to TXT format (Figure 4.1b).

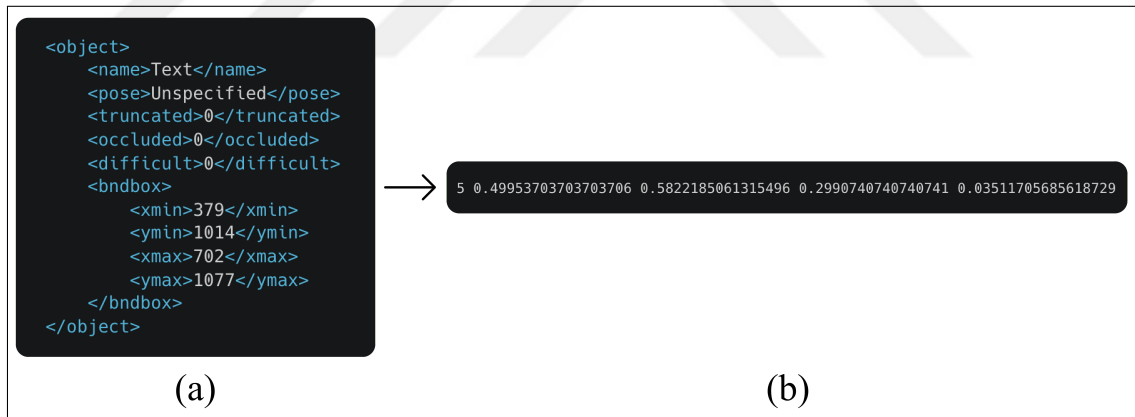


Figure 4.1. An example of conversion from Pascal VOC XML to YOLO TXT

4.2.2. GUI Element Detection Model

The YOLOv5 algorithm is employed to create a custom GUI element detection model. It is deemed appropriate since YOLO family algorithms tend to be fastly deployable - requiring a limited number of libraries and dependency setup; but on the other hand, resulting in a swift and accurate real-time object detection architecture.

Accordingly, initially the YOLOv5 repository, which includes its open-source implementation, is cloned to the computer. Then, the YOLOv5 libraries and dependencies are installed. The YOLOv5 algorithm takes the required configuration data from two text files - namely *train.yaml* and *data.yaml*. The *train.* includes parameters that describe the model architecture. In order to speed up the training process, there are a couple of pre-trained models (*YOLOv5n*, *YOLOv5s*, *YOLOv5m*, *YOLOv5l* and *YOLOv5x*) within the repositories. For the purpose of UI image training, in our study *YOLOv5s*, which is the smallest and fastest model available, is used as the reference starting point. Hence, the pretrained weights of *YOLOv5s* are used as the beginning of the training process to avoid overfitting of the model. In the second configuration file - *data.yaml* - the training, validation, and test set paths on the local PC are provided. Also, a parameter called *nc* is set, which declares the number of classes in the dataset in both of the *train.yaml* and *data.yaml* files. Considering the total number of classes - GUI element types - that are utilized as a part of this training process, the *nc* value is set to 12.

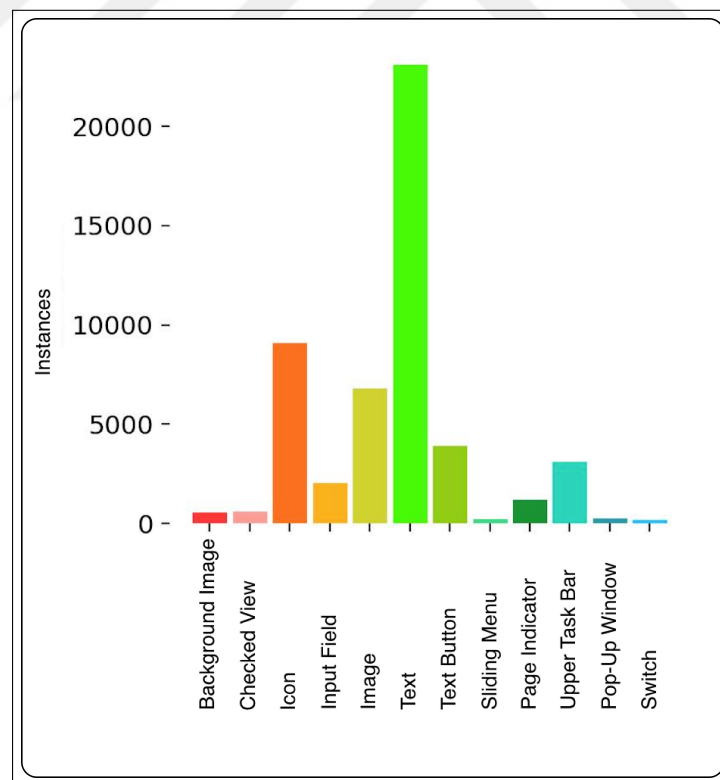
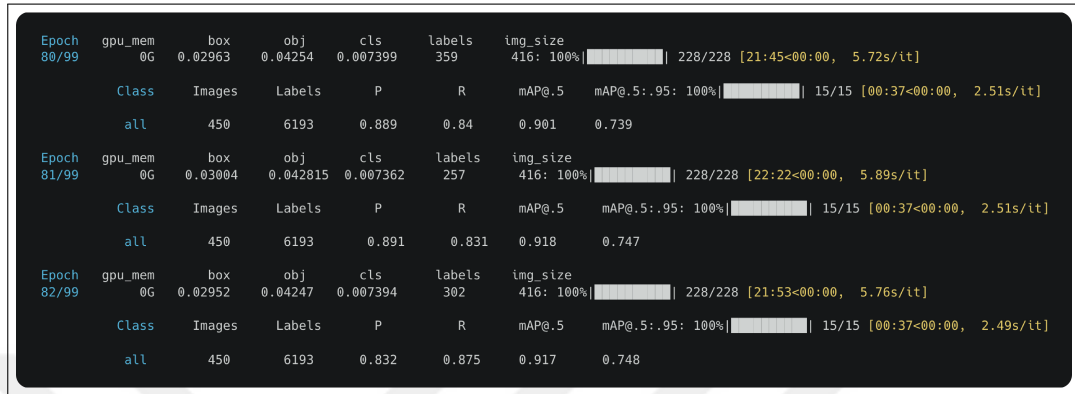


Figure 4.2. Distribution of class labels in the training dataset

After the *yaml* files are configured, the model training phase is initiated. The above figure (Figure 4.2) shows the distribution of the total 50413 annotated GUI instances in 3643 training

images, which are split into 12 classes. The new model is trained (Figure 4.3) for 100 epochs and with a batch size of 16 training images; and as a result, it took 38 hours and 33 minutes to fully train the GUI element detection model.



Epoch	gpu_mem	box	obj	cls	labels	img_size	416: 100%	228/228	[Time, Speed]
80/99	0G	0.02963	0.04254	0.007399	359				[21:45<00:00, 5.72s/it]
	Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95: 100%		15/15 [00:37<00:00, 2.51s/it]
	all	450	6193	0.889	0.84	0.901	0.739		
81/99	0G	0.03004	0.042815	0.007362	257				[22:22<00:00, 5.89s/it]
	Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95: 100%		15/15 [00:37<00:00, 2.51s/it]
	all	450	6193	0.891	0.831	0.918	0.747		
82/99	0G	0.02952	0.04247	0.007394	302				[21:53<00:00, 5.76s/it]
	Class	Images	Labels	P	R	mAP@.5	mAP@.5:.95: 100%		15/15 [00:37<00:00, 2.49s/it]
	all	450	6193	0.832	0.875	0.917	0.748		

Figure 4.3. A snapshot from the training phase

After the training phase is completed, the resulting model is used to detect GUI elements in a given image. This process is achieved by invoking the Python detector script, which is provided as part of the YOLOv5 repository. The detector script is fed with a UI image (Figure 4.4a) and after running the trained model, it infers the relevant UI items and their location on images and saves the annotated image (Figure 4.4b) to a specified destination folder.

4.3. RESULTS & FINDINGS

Since one of the main objectives of this study is to train and evaluate a newly released object detection model, which in this case is used to detect and classify GUI elements in UI images, this chapter initially draws plots using PASCAL VOC (IoU = 0.5) and MS COCO [70] (AP@[0.5:0.05:0.95]) evaluation metrics. The rest of the chapter presents the class base average precision findings and compares them with the benchmark study.

After each epoch during the training process, the validation set is used to evaluate the model. This evaluation is specifically observed to optimize and fine-tune the model hyperparameters for the subsequent epoch of training. Thus, it is ensured that the training proceeds in the right direction without any issues. Accordingly, the total of 450 UI images that are selected as the validation set is utilized to calculate the mAP values at specific IoU thresholds.



Figure 4.4. Running inference on an image using the trained model

As a result, the validation findings show that the model has achieved an mAP (IoU = 0.5) of 93.3% at the last epoch of the training process (Figure 4.5a). Considering the MS COCO dataset challenge, which introduces a new evaluation metric (AP@[0.5:0.05:0.95]), during the training process APs are calculated at 10 IoU thresholds between 0.5 and 0.95 with a step size of 0.05. The average of the 10 calculated APs achieved the mAP of 77.6%. The two calculated mAP results conducted on the validation set show that the model incrementally gives better results in classifying the GUI elements by each epoch.

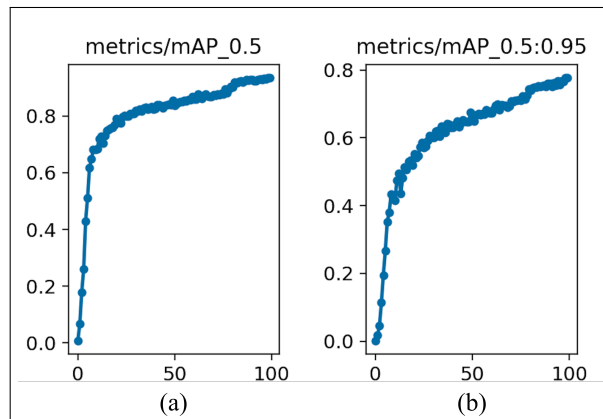


Figure 4.5. Accuracy results of model training on the validation set

The second part of the evaluation entails the comparison of this newly trained model with the existing benchmark study's results. Therefore, after the model training has ended, the Python script, which is provided as part of the YOLOv5 repository, has been deployed on the selected 450 testing UI image dataset. After this process, an AP value is generated per input class which then average is calculated to obtain the overall mAP.

Based on the calculations described above, Table 4.1 depicts the average precision findings for each one of the classes in our prototype and the benchmark study by Bunian et al. As can be clearly seen, the superiority of the model that is trained with the YOLOv5 is apparent to the naked eye. To put in numbers, the average precision of 10 out of 12 classes has yielded better results compared to the Bunian et al. [8] model. Even though the YOLOv5 model has achieved a slightly lower AP score in the *Background Image* and *Sliding Menu* classes than the Bunian et al. [8], it successfully has achieved higher average precision rates in smaller GUI elements, such as *Icon*, *Checked View* and *Page Indicator*. Overall, it is calculated that the proposed YOLOv5 model has outperformed the Bunian et al. [8] model's mAP by 15.69%.

Table 4.1. Average precision (AP) at IoU of 0.5 for each of the 12 class labels on the test set

AP (%) at IoU = 0.5		
Class Label	Bunian et al. [8]	YOLOv5
Background Image	89.33	84.62
Checked View	44.48	67.59
Icon	50.5	89.58
Input Field	78.24	94.57
Image	79.24	83.87
Text	63.99	96.3
Text Button	87.37	98.83
Sliding Menu	100	99.5
Page Indicator	59.37	97.96
Pop-Up Window	93.75	95.52
Switch	80	97.16
Upper Task Bar	90.4	99.49
Overall mAP(%)	76.39	92.08

The findings have been further explored through the confusion matrix (at IoU = 0.5) to evaluate the matching success of the test UI images to the corresponding 12 classes - see Figure 4.6. The figure indicates that all the *Sliding Menu* and *Upper Task Bar* containing images have successfully matched with the correct class. Following this, the second-best performance of matching with the relevant classes has been achieved by the test UI images that contain *Text*, *Text Button*, and *Page Indicator* instances. However, UI images that contain *Checked View* and *Switch* instances have not performed as well. This problem is believed to be related to the size of these UI elements in relation to the image in general. Hence, it can be said that smaller UI elements tend to be harder to detect and locate. Lastly, it has been observed that some akin and look-alike UI types such *Icon* and *Image* are falsely predicted. Specifically, the newly generated model falsely predicted 18% of *Icons* as *Images*; similarly, it falsely predicted 9% of *Images* as *Icons*.

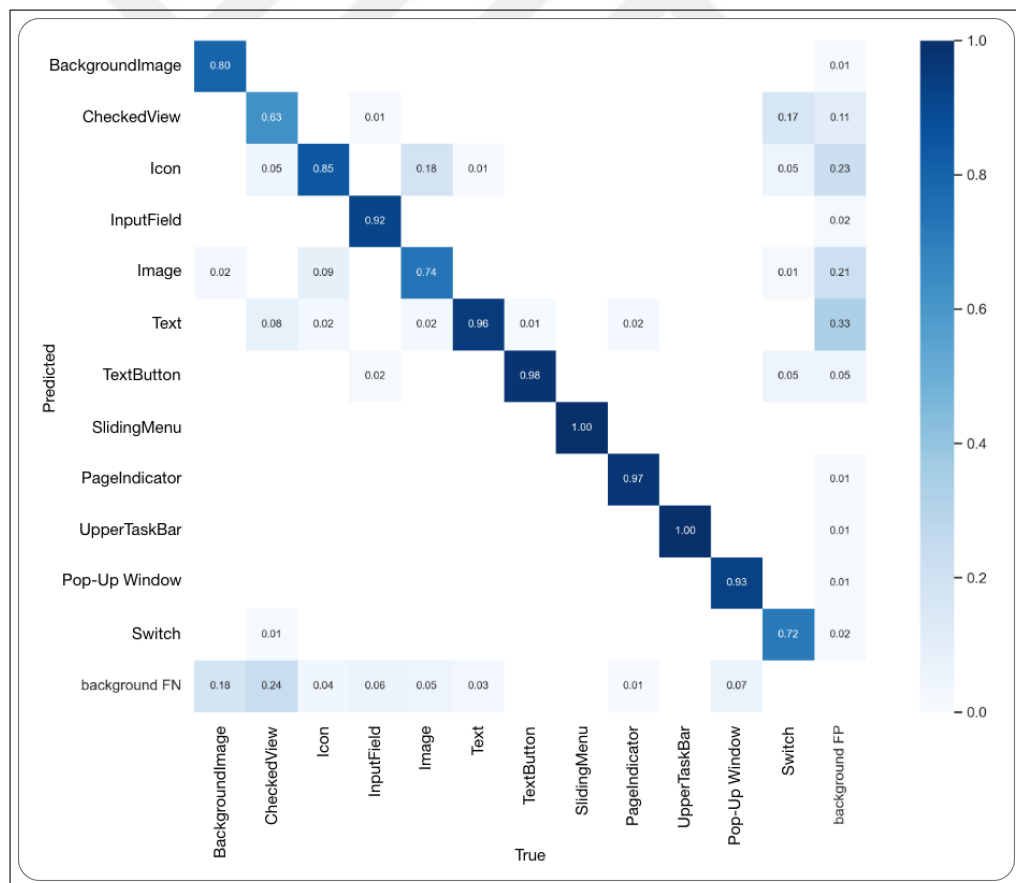


Figure 4.6. Confusion matrix based on 450 test images

5. UI LAYOUT FILE GENERATION

This chapter analyzes the requirements, discusses the system architecture, and details the implementation steps of the proposed UI layout file generation system. Moreover, the evaluation results of the UI layout file system are discussed and elaborated on in this chapter.

5.1. ANALYSIS & DESIGN

To develop a functional UI layout file generation system, several requirements must be met. Firstly, the system must prioritize the accurate identification of GUI components, as this directly affects the system's overall effectiveness. Additionally, the system should be able to properly group and nest GUI elements by acquiring their location. Furthermore, the system must be able to generate consistent UI screens across different screen sizes by taking a mobile UI design as input and using normalized coordinates to obtain the location of the GUI components. The proposed GUI element model can also be utilized in this process, as it naturally provides the coordinate position and type of detected GUI components.

Additionally, the UI layout file generation system must be capable of extracting other attributes of GUI components apart from their types and coordinate positions. For instance, *Input Field* might make use of text attribute to introduce a placeholder or entry. On the other hand, *Icon* and *Background Image* components might utilize vector images or illustrations to carry out appealing UI screens. Last but not least, GUI elements might have a custom color attribute to create balance and consistency through the components and emphasize particular aspects of a design.

Bearing in mind all the above, the system must also be versatile. The identified GUI components should be adapted into the equivalent UI elements in iOS and Android platforms. In addition, the system must be able to generate UI files for both platforms. These files can be used to visualize the output of the system and also evaluate its performance.

Being aware of the aforementioned requirements, it is important to note that the system that would carry out the implementation of the UI layout file generation system should have enough resources to manage the development phase. It should be able to store the artifacts generated

during the development process. Furthermore, it should be capable of elaborating on the outcomes and results of the system. Additionally, the system must be developed targeting Mac OS, as Mac computers are the most suitable platform to develop apps for iOS and it also supports Android IDEs. Last but not least, the Python programming language should be used to perform image processing and machine learning tasks as it provides simplicity and consistency, and access to fundamental libraries and frameworks.

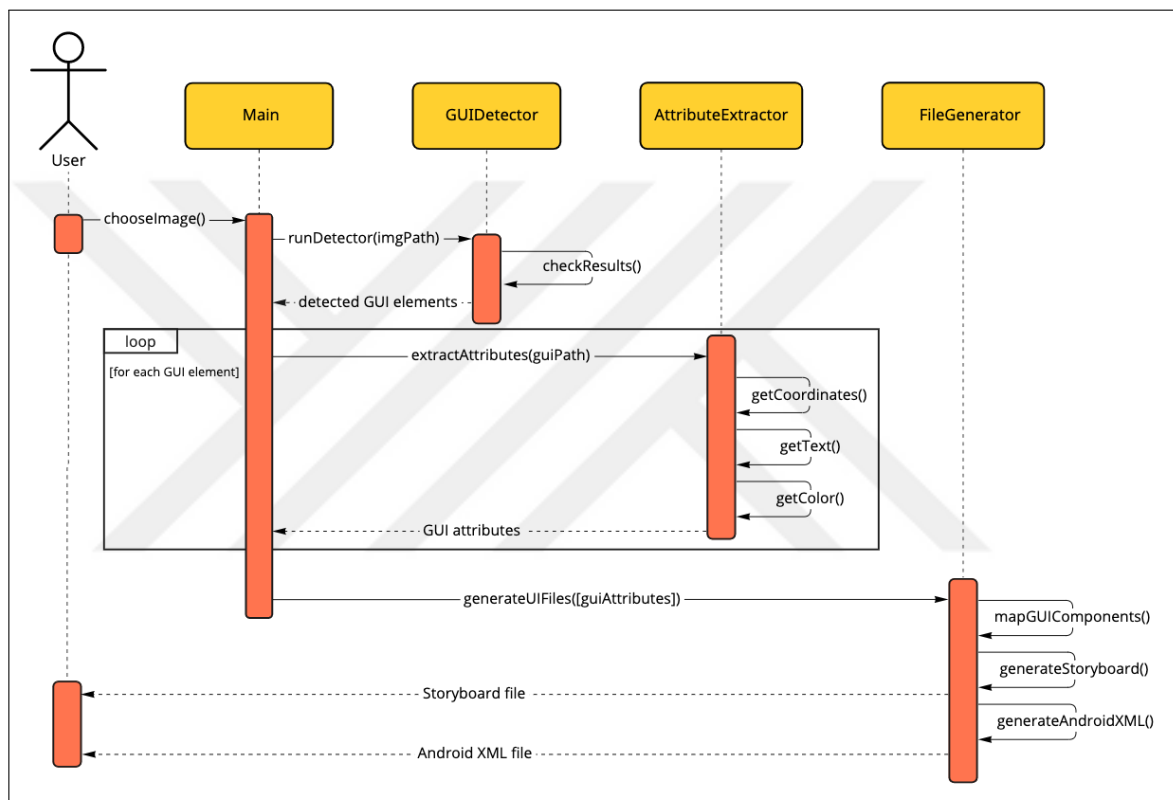


Figure 5.1. Sequence diagram of the proposed system

As depicted in Figure 5.1, the sequence diagram of the system is visualized and the interactions among the group of objects are described. There are 4 objects — *Main*, *GUIDetector*, *AttributeExtractor*, and *FileGenerator* — to represent the activities of the system.

Once the *User* selects an image of mobile UI design, the *Main* runs to start the GUI detection phase of the UI layout file generation process. Then *GUIDetector* processes the image, extracts the GUI components, and returns an array of the detected GUI elements to the *Main*. Once the *Main* receives the detected GUI elements, the *Main* then sends each GUI element to *AttributeExtractor* to obtain more detailed specifications. Following this, *AttributeExtractor* sends back the identified GUI attributes to *Main*. Subsequently, *Main* sends the array of GUI

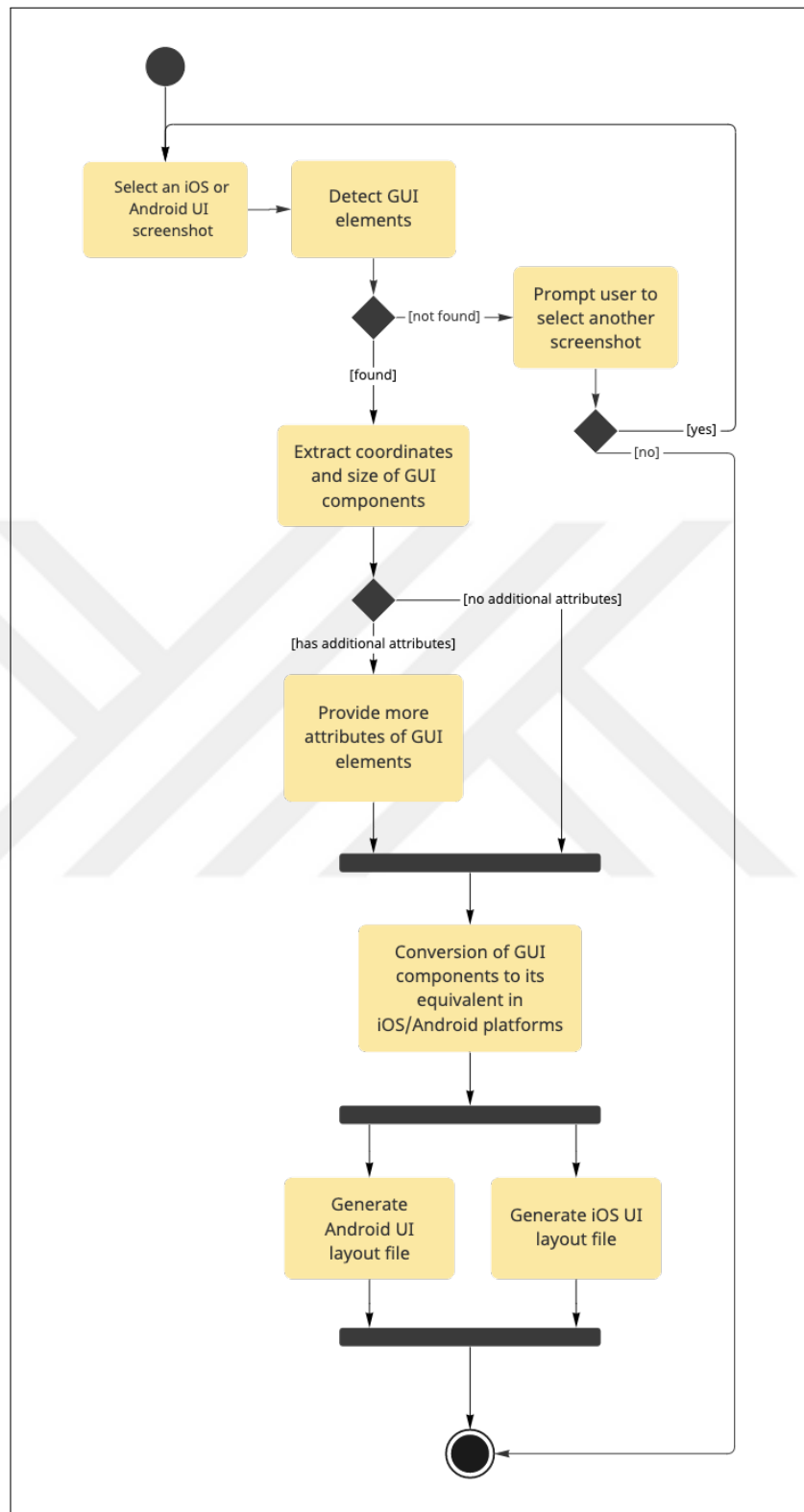


Figure 5.2. Activity diagram of the proposed system

attributes to *FileGenerator*, and then *FileGenerator* performs UI layout file generation for both iOS and Android platforms. In conclusion, the *FileGenerator* then sends the generated UI layout files back to the *User*.

Moreover, Figure 5.2 depicts the flow of activities within the system to generate UI layout files. In this process, the user first selects an image that could be an iOS or Android UI design. The system then utilizes the selected image to detect GUI components, it checks the number of GUI components and prompts the user if nothing is detected. If one or more GUI elements are detected, the system obtains their coordinates and sizes, then converts the detected GUI elements and their properties to their equivalent in iOS and Android platforms. Finally, the system automatically generates alternative layout files (iOS and Android) based on predefined design rules.

5.2. IMPLEMENTATION

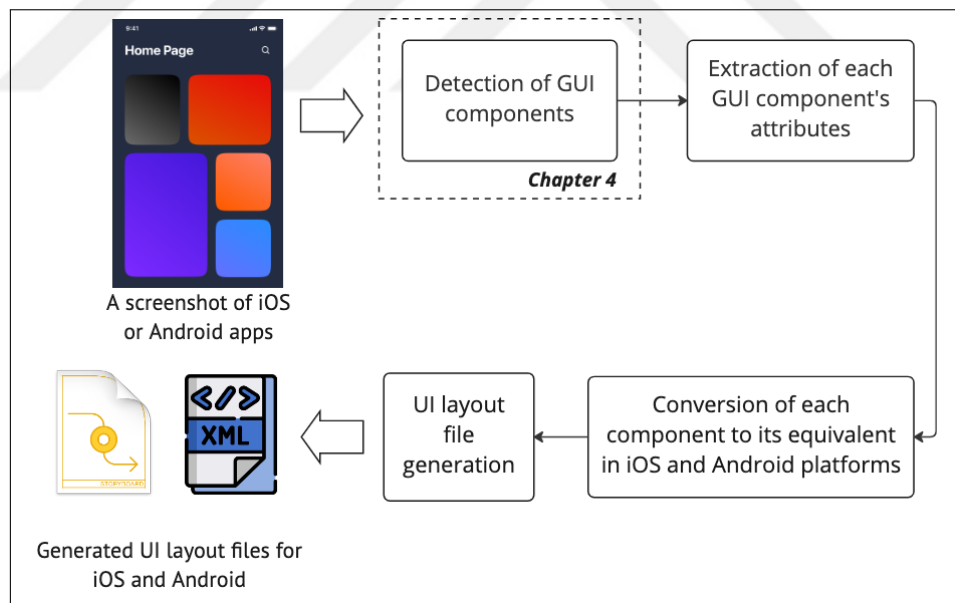


Figure 5.3. Overview of the UI layout file generation system

The implementation phase of the UI layout file generation system is a multi-step process Figure 5.3. The example screenshot is used to guide the step-by-step explanation of the overall process of the UI layout file generation system. To begin with, the system takes an iOS or Android screenshot as input. This input is fed into the proposed GUI element detection model — see Chapter 4 — and the model runs an inference on the image to perform GUI

element classification and localization. As a result, an annotated image is achieved along with GUI component locations and their types.

5.2.1. Extracting GUI components

The repository that is used to employ the YOLOv5 algorithm provides a text file along with the annotated image to save the specifications of the identified GUI elements. Within this file, each line represents a detected GUI element and contains the GUI component's type (class number), center coordinates (x and y values), height, and width respectively (Figure 5.4). The class numbers are zero-indexed and the indexes of the classes are used as an identifier for the class labels in the GUI element detection model. The indexes of class labels are designated in *data.yaml* (see Table A.1).

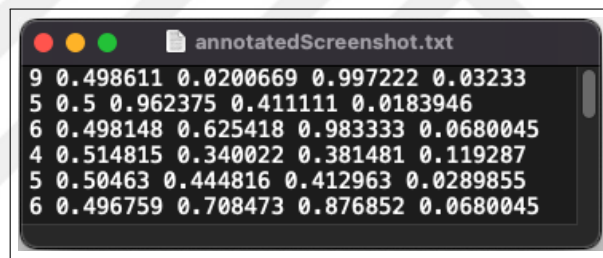


Figure 5.4. The annotation file for the input screenshot

On the other hand, the coordinates, width, and height which are normalized by the dimensions of the UI screenshot can lie between 0.0 and 1.0. Normalized annotations make it easier to work with scaled or stretched images. The annotation file contains important information such as the coordinates, width, and height of GUI components in the image. A Python script is used to leverage these values to perform various tasks. This script crops and saves each GUI component in corresponding folders. The script is designed to crop and save each GUI component in the appropriate folder. When applied to the input screenshot, it generates a set of 6 cropped GUI components. These components are illustrated in Figure 5.5.

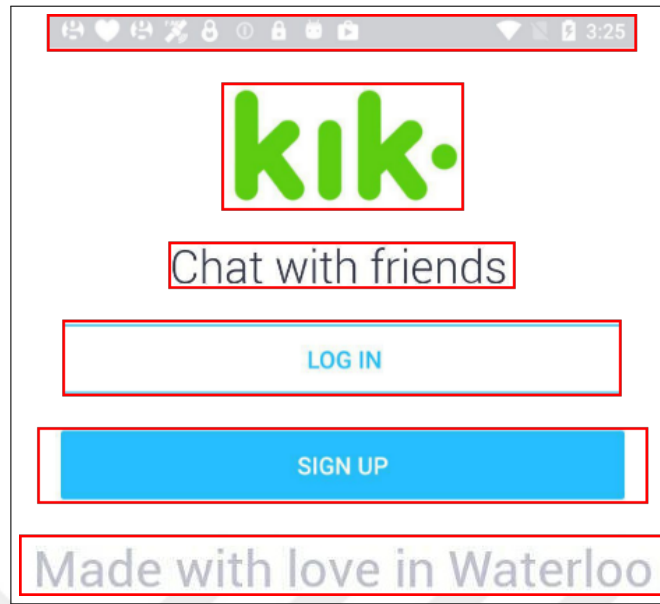


Figure 5.5. Cropped GUI components from input screenshot

5.2.2. Detecting Text and Color Attributes of GUI Elements

The implementation of the proposed system is followed by extracting additional attributes of the cropped GUI components. First off, a Python script is developed that employs pytesseract [71] and OpenCV libraries to extract text from the GUI components. pytesseract is an OCR tool for Python to recognize text from the GUI components. Also, OpenCV supports a multitude of image processing algorithms such as color conversion, thresholding, dilation, and contouring. These algorithms are used to determine text area and thereby increase the accuracy of text detection. The script only runs for GUI element types such as *Text*, *Text Button* or *Input Field*, as they are the types of elements that can be extracted as text. The script (Algorithm B.1) starts converting RGB image (Figure 5.6a) to grayscale image (Figure 5.6b). Then, OTSU threshold [72] which transforms the grayscale image to a binary image to increase the probability of text recognition is applied to the converted image (Figure 5.6c).

Following this, the contour algorithm is used to detect the boundary edges of the text blocks of the binary image. Loop through each contour, bounding boxes are drawn around each text block (Figure 5.6d). Lastly, each bounding box is treated as an individual rectangular region and fed into the pytesseract algorithm to obtain the text from the image. The extracted texts are saved in a text file (Figure 5.6e).

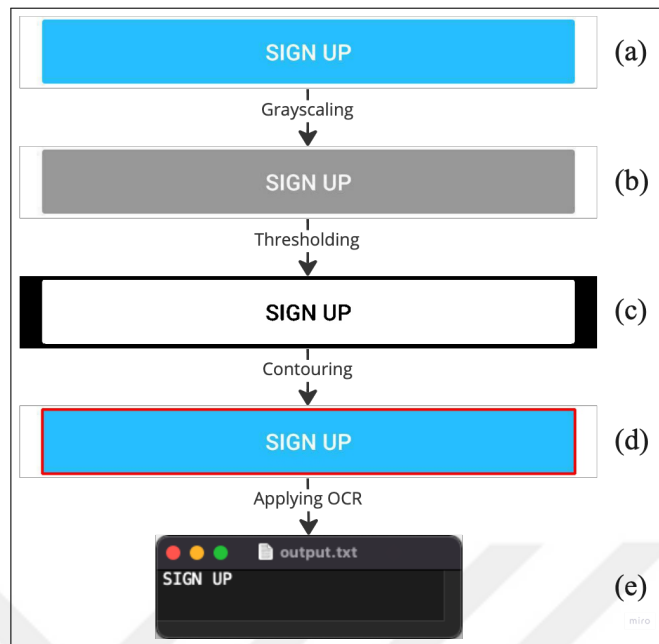


Figure 5.6. Steps of extracting text in GUI components

The color attribute is also considered one of the fundamental properties of a GUI component since it can be used as a text color or background color (Figure 5.7a). *colorthief* [73] which is a Python library for grabbing the color palette from an image is used to obtain the dominant colors in the GUI components. The most dominant color is determined as the background color, whereas, the second most one is obtained as the text color (Figure 5.7b). The colors are saved as RGB codes in a text file (Figure 5.7c).

5.2.3. Generation of the UI Layout Files

The UI layout file generation process starts with the mapping process of the GUI component types to its corresponding UI element in iOS and Android. As it is shown in Table 5.1, most of the GUI components have their equivalent in both platforms. However, if the types of GUI components do not match any of the supported UI elements in one of these platforms, the mapping process ignores these GUI components. For instance, a sliding menu component is usually hidden off-screen and can be revealed by the user via a click of a button or icon. When the sliding menu is invoked, it slides in from the left side of the screen and overlays the main content. Android development provides a panel component that is called *NavigationDrawer* to create a sliding menu easily. However, in iOS, a sliding menu is not a default component.

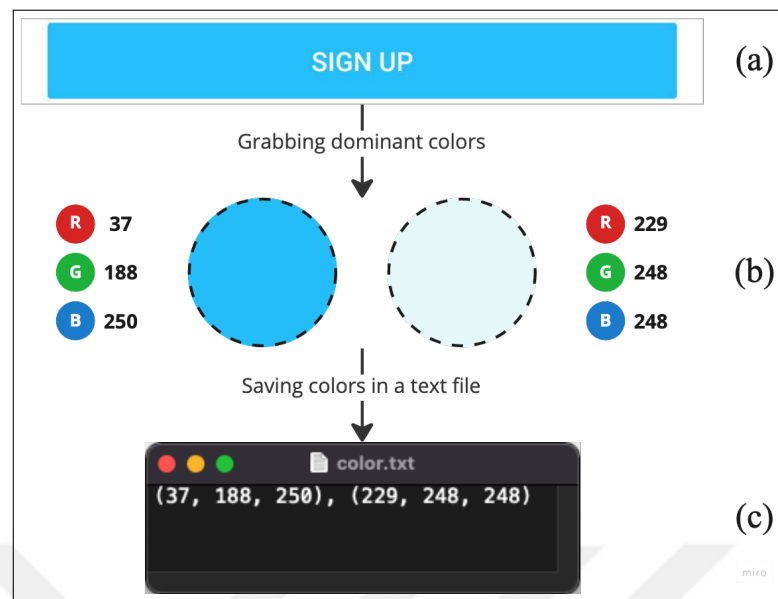


Figure 5.7. Detecting dominant colors in the GUI element

It needs to be implemented explicitly with custom views and gestures.



Figure 5.8. iOS code snippet generation

As stated before, YOLOv5 detects the bounding boxes of the GUI components in normalized coordinates (see Figure 5.4). In order to specify the frame of the GUI component in the UI layout files, the normalized coordinates are converted into pixel coordinates with the help of a Python script. This script takes normalized coordinates and the size of the input image and calculates pixel coordinates and the size of the GUI component.

The process is followed by the structuring of the UI layout files for the iOS and Android platforms. As a part of the iOS development lifecycle, when a new iOS project is created in Xcode, the project may also include a default storyboard that defines an empty scene (or view controller) and some metadata about the layout file. Therefore, the default setup of the storyboard is used as a starting point for the UI layout generation. Furthermore, a python

Table 5.1. Android and iOS equivalents of GUI components detected by the GUI element detection model

Class Label	Android	iOS
Background Image	ImageView	UIImageView
Checked View	CheckBox	UISwitch
Icon	ImageView	UIImageView
Input Field	EditText	UITextField
Image	ImageView	UIImageView
Text	TextView	UILabel
Text Button	Button	UIButton
Sliding Menu	SlidingDrawer	-
Page Indicator	-	UIPageControl
Pop-Up Window	PopupWindow	-
Switch	Switch	UISwitch
Upper Task Bar	-	-

script is developed to create and modify the storyboard file. The script sets the scene to freeform and resizes it as same as the input image. Then, it produces the code fragments (Figure 5.8) for each GUI element along with its properties. A code fragment is created by using multiline string variables that correspond to different types of iOS UI components. These variables take parameters to update their fields with attributes such as color, text, position, width, and height. Once the code fragments are generated, they are added to the storyboard file.

On the other hand, XML (Extensible Markup Language) files are used to create the layout for an Android app's user interface. It is used to specify the layout of the visual components of an app, such as buttons, text fields, and image views. A similar python script, used to generate code blocks for iOS UI components, as shown in Figure 5.9, has been developed to accomplish the same task for Android UI components. Consequently, the script completes its run and saves generated UI layout files of both platforms — see Figure D.1 and Figure D.2 — in the designated path in the computer.

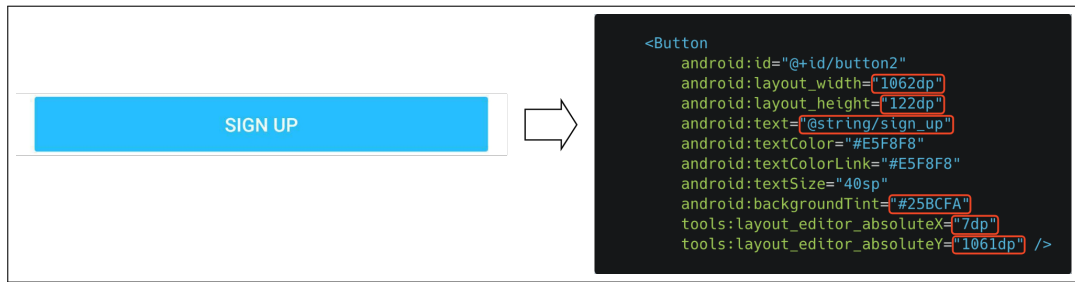


Figure 5.9. Android code snippet generation

5.3. RESULTS & FINDINGS

The purpose of the proposed system is to achieve the automation of the UI layout file generation. To address the system's performance, multiple test metrics have been utilized and conducted. To begin with, 5 different test images (Figure 5.10) from the VINS dataset have been randomly selected as a representative sample to evaluate the performance of the proposed algorithm. The VINS dataset is used as the source of test images since it is also used in the first phase of the thesis, ensuring consistency in the evaluation process. This approach allows for a fair and unbiased assessment of the algorithm's capabilities, as the sample images are randomly chosen from the dataset used in the study rather than cherry-picked. Then, each test image is employed in the system and processed through the aforementioned implementation steps. As a result, the system generates the UI layout files for iOS and Android platforms respectively.

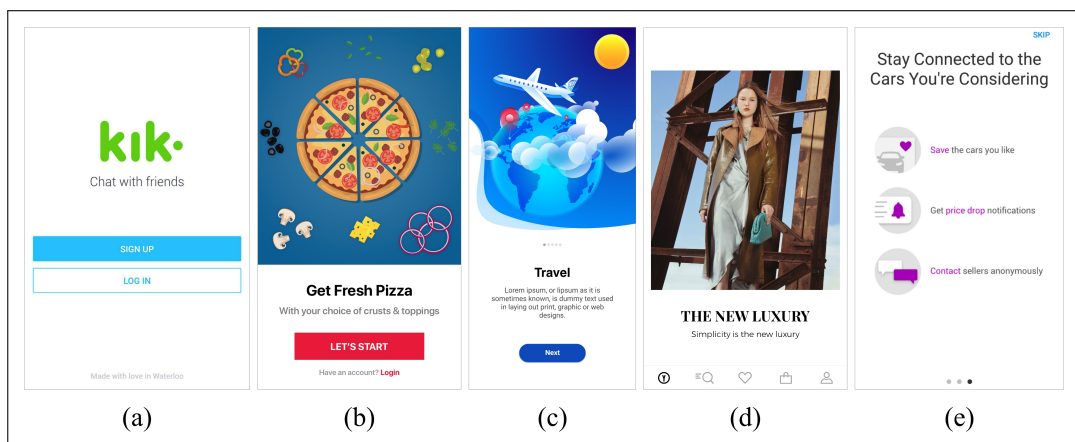


Figure 5.10. Selected test images

In addition, UI layout files demonstrate a graphical representation of the scenes when they

are opened with a compatible integrated development environment (IDE) such as Xcode and Android Studio. Bearing in mind that, the screenshot of the iOS and Android scenes are captured from the UI layout files to compare the similarity against the test images.

5.3.1. Structural Similarity Index Measure (SSIM)

To evaluate the similarity between each test image and its generated iOS and Android UI designs, SSIM can be a useful tool for comparison. SSIM is a suitable metric for this task as it measures structural similarity, which is important for UI design comparison as it takes into account the structural information of images, such as the arrangement of pixels, rather than just comparing pixel values, making it more robust to changes in lighting and other factors that can affect the appearance of an image. To achieve this, the *sewar* [74] which is a Python library that serves a multitude of functions to compute several image similarity metrics, is used to calculate SSIM.

The code snippet in Figure 5.11 illustrates the use of the SSIM function to compare a test image with the generated iOS and Android screenshots. It loads the three images - a test image, an iOS UI design, and an Android UI design - then calculates the SSIM score between the test image and the iOS-Android screenshots. The score is then displayed on the console (Figure 5.11). The results of the comparison show that the system produces an average similarity score of 0.731 for iOS designs and 0.703 for Android designs (Table 5.2).

```
from sewar.full_ref import ssim
import cv2

testImage1 = cv2.imread("testImage1.png")
iosImg = cv2.imread("iosImg1.png")
androidImg = cv2.imread("androidImg1.png")

print("SSIM (testImage1 - iosImg): ", ssim(testImage1, iosImg))
print("SSIM (testImage1 - androidImg): ", ssim(testImage1, androidImg))
```

Figure 5.11. SSIM calculation script

The structural similarity score can be affected by a number of factors. For instance, the system's ability to accurately detect and extract GUI elements plays a vital role in the final

generated UI designs, and if the system is not able to accurately detect and extract the elements, it could lead to lower similarity scores — such as *Image b* and *Image d*. Additionally, the design rules and constraints used in the UI layout generation process can also affect the similarity of the generated designs to the original test image, as different rules and constraints can lead to variations in the final designs. Furthermore, the quality and resolution of the test image, as well as any variations or inconsistencies in the image, can also affect the similarity score. Lastly, the variation of the design styles and elements used in the iOS and Android platforms can also lead to different similarity scores between the two.

Table 5.2. SSIM results between test images and its generated iOS & Android UI designs

SSIM score		
Image	Android	iOS
a	0.819	0.832
b	0.565	0.630
c	0.752	0.766
d	0.547	0.581
e	0.833	0.847
Average	0.703	0.731

5.3.2. Pixel-based Similarity

The system's ability to accurately recreate pixel-based resemblance between test images and generated UI designs is evaluated by using the *blink-diff* library. The library allows the identification of differences between two screenshots by comparing them pixel by pixel. The library's API takes the first test image (Figure 5.12a), compares it to the generated UI design (Figure 5.12c), and highlights any differences in red in the output image (Figure 5.12b). Furthermore, the system's ability to replicate pixel-based similarities between the test images and generated UI designs was conducted. The results indicate that, on average, the system was able to replicate 83.41% of the pixel-based details for designs intended for the iOS platform and 81.29% for designs intended for the Android platform, when considering the given test images (Table 5.3). It can be inferred that the system has effectively captured the visual details and nuances of the test images in the generated designs. This highlights the

system's capability to produce high-fidelity UI designs that closely resemble the original test images.

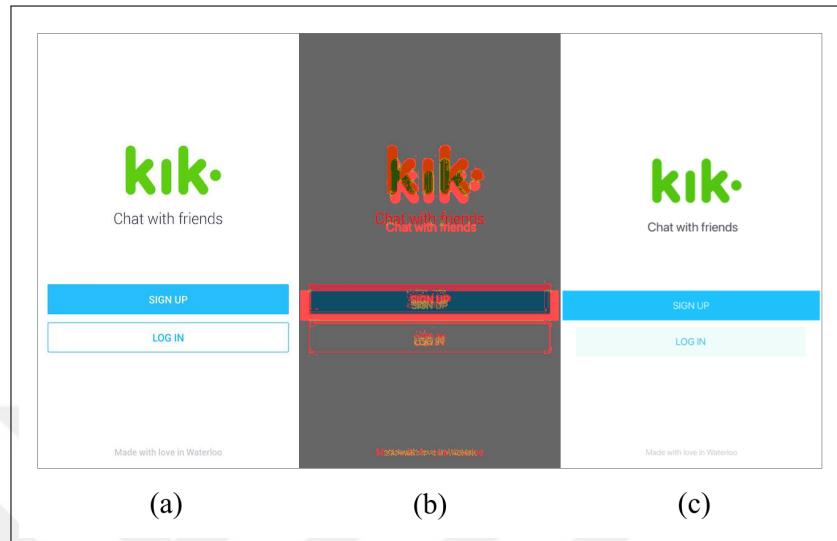


Figure 5.12. Pixel-based comparison of the first test image and its generated iOS design

Lastly, there can be several reasons for the difference between iOS and Android results. One possibility is that the system may have performed better on the iOS designs due to the more consistent design guidelines and layout patterns commonly used in iOS development. Additionally, the system may have more difficulty replicating the pixel-based details for Android designs due to the wider range of devices and screen sizes are used in the Android ecosystem. Furthermore, the dataset and the way the images are captured and pre-processed before the comparison can also be a reason for the difference between the results.

Table 5.3. Pixel-based similarity between test images and its generated iOS & Android UI designs

Pixel-based similarity (%)		
Image	Android	iOS
a	87.16	91.93
b	66.2	69.3
c	81.11	81.24
d	77.13	79.09
e	94.86	95.51
Average (%)	81.29	83.41

6. CONCLUSION & FUTURE WORK

This study is proposed to aid in streamlining the implementation of user interfaces in the mobile application development process. Also, it is introduced to create optimized layouts for different devices and screen sizes, improving consistency and ease of maintenance, ultimately leading to a better user experience. In this sense, this study is divided into two phases. The first phase focuses on the development of a custom object detection model for identifying GUI elements. The second phase concerns the creation of a the UI layout file generation system, which utilizes the custom GUI element detection model developed in the first phase.

First and foremost, this study proposes, trains, and develops a GUI element detection model by making use of the YOLOv5 algorithm and VINS dataset. The original VINS dataset contains 4800 UI images and their annotation files, which is reduced to 4543 images to make it comparable with the Bunian et al. [8] as a part of this study. Accordingly, the total of 4543 images are split into 3 subsets for training, validation, and testing purposes. Furthermore, the accompanying VINS dataset annotation files are converted into YOLO TXT format. The model is trained using the newly proposed YOLOv5 algorithm that employs the pre-selected training dataset. Then in line with the benchmark study, 450 images are used for validation and the same number of images for testing. Accordingly, it is observed that the model has given 15.69% better mAP than the model developed and proposed in the Bunian et al. [8] study. Overall, the comparison showed that the newly created model has performed better than the benchmark study in 10 out of 12 classes.

Based on the experience gained throughout the evaluations, it is foreseen that if the classes used for training are more evenly distributed the final results tend to be much more accurate. Even though the newly generated model outperforms the set benchmark study findings, there is still room for improvement in the detection of small GUI elements. It is believed that this could be achieved by tiling the dataset [75] or using a different YOLO-based algorithm specifically developed for small object detection. The success of this phase would inevitably lead to the improvement of the next phase which involves automated UI file generation and UI design regeneration for multiple mobile platforms.

Moreover, Wang et al. [76] proposed YOLOv7 which is a variant of the YOLO algorithm

known for its speed and accuracy in object detection tasks. YOLOv7 uses a CNN architecture called SPP-YOLO (Spatial Pyramid Pooling YOLO) for object detection. The SPP-YOLO architecture consists of a series of convolutional layers followed by an SPP layer, which helps to preserve spatial information and reduce the computational cost of the network. One key advantage of YOLOv7 is its efficiency. It is able to process images at a high frame rate, making it suitable for real-time object detection applications such as video surveillance and autonomous vehicles. YOLOv7 is also relatively easy to train and fine-tune, making it a popular choice for many object detection tasks. At the time the first part of this study was published [77], the YOLOv7 had not been released yet. For this reason and the advantages mentioned above, it is worth trying to train the GUI detection model via the YOLOv7 algorithm and elaborate on whether it outperforms YOLOv5 or not.

The second part of this thesis proposes and develops a system that engages the GUI element detection model achieved in the first phase to generate UI layout files for a given mobile UI design. Accordingly, the input image is fed into the GUI element detection model. Then, the model delivers an annotated image along with GUI components. After that, each detected GUI component has been processed to extract its additional features such as color, text, size, and position. Following this, a Python script is developed to produce code fragments that include UI representation of each detected GUI element. In consequence, the proposed system is evaluated by using image similarity metrics. The test results show that the proposed system has produced very similar UI designs of iOS and Android platforms for given test images. Such that, the generated UI designs have been evaluated in terms of structural similarity and pixel-based evaluation metrics. On average, it is obtained that the system has generated iOS and Android UI designs with a structural similarity score of 0.731 and 0.703 respectively. Moreover, pixel-based similarity has been assessed in the proposed system. It is observed that the generated iOS and Android UI designs have achieved 83.41% and 81.29% on average pixel-based similarity for given test images.

As it is demonstrated, the evaluations show that the effectiveness and the success of the proposed UI layout file can be enhanced and there is still room for improvement. To suggest some directions and venues; for instance, the proposed system can be improved to lay out the UI components by using specific tools of iOS and Android platforms. In that manner, *UIStackView* class can be used to provide a streamlined interface for laying out a collection

of views in iOS. It can be configured to distribute the views evenly across its available space or to allocate specific amounts of space to each view. Likely, in Android, *RelativeLayout* and *LinearLayout* classes can be used for the very same purpose.

Moreover, the system can be extended to achieve more visual attributes from input UI designs. For example, in its current state, the system is not capable of detecting font style and size at the moment. Also, the system assumes the input UI designs always have a white background. Moreover, this system is not capable of detecting font types of GUI components. So, the default font types are used in the generation of UI layout files on both platforms. By addressing the issues mentioned above, conducting further research in these areas to solve these drawbacks could have a significant impact, and ultimately help broaden the reach of the system's methodology.

REFERENCES

1. Your smartphone is millions of times more powerful than the Apollo 11 guidance computers [cited 2022 15 January]. Available from: <https://www.zmescience.com/science/news-science/smartphone-power-compared-to-apollo-432>.
2. Number of apps available in leading app stores as of 1st quarter 2021 [cited 2022 30 January]. Available from: <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores>.
3. Revenue of mobile apps worldwide 2017-2025, by segment [cited 2022 2 February]. Available from: <https://www.statista.com/forecasts/1262892/mobile-app-revenue-worldwide-by-segment>.
4. Chen J, Xie M, Xing Z, Chen C, Xu X, Zhu L, et al. Object detection for graphical user interface: Old fashioned or deep learning or a combination? *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020:1202-14.
5. Xie M, Feng S, Xing Z, Chen J, Chen C. UIED: a hybrid tool for GUI element detection. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020:1655-9.
6. Torbert S. *Applied computer science*. Springer; 2016.
7. Suzuki S, et al. Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing*. 1985;30(1):32-46.
8. Bunian S, Li K, Jemmali C, Hartevelde C, Fu Y, Seif El-Nasr MS. Vins: Visual search for mobile user interface design. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021:1-14.
9. Chen C, Feng S, Xing Z, Liu L, Zhao S, Wang J. Gallery dc: Design search and knowledge discovery through auto-created gui component gallery. *Proceedings of the ACM on Human-Computer Interaction*. 2019;3(CSCW):1-22.

10. Su T, Meng G, Chen Y, Wu K, Yang W, Yao Y, et al. Guided, stochastic model-based GUI testing of Android apps. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017:245-56.
11. Chen S, Fan L, Su T, Ma L, Liu Y, Xu L. Automated cross-platform GUI code generation for mobile apps. *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. IEEE. 2019:13-6.
12. Canny Edge Detection [cited 2023 5 January]. Available from: https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html.
13. Eroding and Dilating [cited 2023 5 January]. Available from: https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html.
14. Ge X. Android GUI search using hand-drawn sketches. *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019:141-3.
15. Cheng J, Tan D, Zhang T, Wei A, Chen J. YOLOv5-MGC: GUI Element Identification for Mobile Applications Based on Improved YOLOv5. *Mobile Information Systems*. 2022;2022.
16. Arthur D, Vassilvitskii S. k-means++: The advantages of careful seeding. Stanford; 2006.
17. Zhu P, Li Y, Li T, Yang W, Xu Y. GUI Widget Detection and Intent Generation via Image Understanding. *IEEE Access*. 2021;9:160697-707.
18. Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*. 2014.
19. Beltramelli T. pix2code: Generating code from a graphical user interface screenshot. *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 2018:1-6.
20. Aşıroğlu B, Mete BR, Yıldız E, Nalçakan Y, Sezen A, Dağtekin M, et al. Automatic HTML code generation from mock-up images using machine learning techniques. *2019*

Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science (EBBT). IEEE. 2019:1-4.

21. Contours: Getting Started [cited 2023 5 January]. Available from: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html.
22. Nguyen TA, Csallner C. Reverse engineering mobile application user interfaces with remaui (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015:248-59.
23. Moran K, Bernal-Cárdenas C, Curcio M, Bonett R, Poshyvanyk D. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*. 2018;46(2):196-221.
24. Chen S, Fan L, Chen C, Su T, Li W, Liu Y, et al. Storydroid: Automated generation of storyboard for Android apps. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019:596-607.
25. Yun YS, Jung J, Eun S, So SS, Heo J. Detection of gui elements on sketch images using object detector based on deep neural networks. *International Conference on Green and Human Information Technology*. Springer. 2018:86-90.
26. Redmon J, Farhadi A. YOLO9000: better, faster, stronger. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017:7263-71.
27. Lin T. labelImg. GitHub. 2015. <https://github.com/heartexlabs/labelImg>.
28. Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L. Imagenet: A large-scale hierarchical image database. *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009:248-55.
29. Bhambure S, Chavan R, Deolekar A, Dhanawade K, Nitnaware P. UI Code Generation using Deep learning. *PCE JCE_*. 2020:105-9.
30. Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*. 2014.

31. Chollet F, et al.. Keras. GitHub. 2015. <https://github.com/fchollet/keras>.
32. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al. {TensorFlow}: a system for {Large-Scale} machine learning. *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016:265-83.
33. Feng S, Jiang M, Zhou T, Zhen Y, Chen C. Auto-Icon+: An Automated End-to-End Code Generation Tool for Icon Designs in UI Development. *ACM Transactions on Interactive Intelligent Systems (TiiS)*. 2022.
34. Müllner D. Modern hierarchical, agglomerative clustering algorithms. *arXiv preprint arXiv:11092378*. 2011.
35. Chen C, Su T, Meng G, Xing Z, Liu Y. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. *Proceedings of the 40th International Conference on Software Engineering*. 2018:665-76.
36. Huang X, Liao F. Automatically Generating Codes from Graphical Screenshots Based on Deep Autocoder. *arXiv preprint arXiv:200702272*. 2020.
37. OpenCV About [cited 2022 13 February]. Available from: <https://opencv.org/about>.
38. Apple Developer Vision [cited 2022 16 February]. Available from: <https://developer.apple.com/documentation/vision>.
39. Google Colab [cited 2022 1 January]. Available from: <https://colab.research.google.com>.
40. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*. 2019;32.
41. Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:14085093*. 2014.
42. Ferrucci DA. Introduction to “this is watson”. *IBM Journal of Research and Development*. 2012;56(3.4):1-1.
43. Cloud Vision API [cited 2022 3 November]. Available from: <https://cloud.google.com/>

vision.

44. Amazon Rekognition Documentation [cited 2022 2 November]. Available from: <https://docs.aws.amazon.com/rekognition/index.html>.
45. Dalal N, Triggs B. Histograms of oriented gradients for human detection. *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*. vol. 1. Ieee. 2005:886-93.
46. Hochreiter S, Schmidhuber J. Long short-term memory. *Neural computation*. 1997;9(8):1735-80.
47. Sherstinsky A. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena*. 2020;404:132306.
48. Zhiqiang W, Jun L. A review of object detection based on convolutional neural network. *2017 36th Chinese control conference (CCC)*. IEEE. 2017:11104-9.
49. Girshick R, Donahue J, Darrell T, Malik J. Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014:580-7.
50. Girshick R. Fast r-cnn. *Proceedings of the IEEE international conference on computer vision*. 2015:1440-8.
51. Ren S, He K, Girshick R, Sun J. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*. 2015;28.
52. Liu W, Anguelov D, Erhan D, Szegedy C, Reed S, Fu CY, et al. Ssd: Single shot multibox detector. *European conference on computer vision*. Springer. 2016:21-37.
53. Redmon J, Divvala S, Girshick R, Farhadi A. You only look once: Unified, real-time object detection. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016:779-88.
54. Russell BC, Torralba A, Murphy KP, Freeman WT. LabelMe: a database and web-based tool for image annotation. *International journal of computer vision*. 2008;77(1):157-73.

55. ai Corporation C. Computer Vision Annotation Tool (CVAT). GitHub. 2022. <https://github.com/opencv/cvat>.
56. Corporation M. VoTT. GitHub. 2018. <https://github.com/Microsoft/VoTT>.
57. Dutta A, Zisserman A. The VIA annotation software for images, audio and video. *Proceedings of the 27th ACM international conference on multimedia*. 2019:2276-9.
58. Lin T. Labelbox. GitHub. 2018. <https://github.com/Labelbox/labelbox>.
59. SuperAnnotate [cited 2022 17 October]. Available from: <https://www.superannotate.com/annotation-tool>.
60. Supervisely. Supervisely. GitHub. 2017. <https://github.com/supervisely/supervisely>.
61. Deka B, Huang Z, Franzen C, Hibschan J, Afegan D, Li Y, et al. Rico: A mobile app dataset for building data-driven design applications. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 2017:845-54.
62. Leiva LA, Hota A, Oulasvirta A. Enrico: A dataset for topic modeling of mobile UI designs. *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services*. 2020:1-4.
63. What is Uplabs? [cited 2022 18 February]. Available from: <https://www.uplabs.com/faq>.
64. Sermuga Pandian VP, Suleri S, Jarke PDM. UISketch: a large-scale dataset of UI element sketches. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021:1-14.
65. Cocoa Application Competencies for iOS [cited 2022 3 December]. Available from: <https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html>.
66. Understanding the 3 most common loss functions for Machine Learning Regression [cited 2022 12 December]. Available from: <https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression>.
67. Hore A, Ziou D. Image quality metrics: PSNR vs. SSIM. *2010 20th international*

- conference on pattern recognition*. IEEE. 2010:2366-9.
68. Normalized 2-D cross-correlation [cited 2022 12 December]. Available from: <https://www.mathworks.com/help/images/ref/normxcorr2.html>.
 69. Erz M. blink-diff. GitHub. 2014. <https://github.com/yahoo/blink-diff>.
 70. Lin TY, Maire M, Belongie S, Hays J, Perona P, Ramanan D, et al. Microsoft coco: Common objects in context. *European conference on computer vision*. Springer. 2014:740-55.
 71. Pytesseract [cited 2022 28 November]. Available from: <https://pypi.org/project/pytesseract/>.
 72. Otsu N. A threshold selection method from gray-level histograms. *IEEE transactions on systems, man, and cybernetics*. 1979;9(1):62-6.
 73. Dhakar L. colorthief. GitHub. 2013. <https://github.com/lokesh/color-thief/>.
 74. Khalel A. sewar. GitHub. 2018. <https://github.com/andrewekhalel/sewar>.
 75. Ozge Unel F, Ozkalayci BO, Cigla C. The power of tiling for small object detection. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2019:0-0.
 76. Wang CY, Bochkovskiy A, Liao HYM. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:220702696*. 2022.
 77. Altinbas MD, Serif T. GUI Element Detection from Mobile UI Images Using YOLOv5. *International Conference on Mobile Web and Intelligent Information Systems*. Springer. 2022:32-45.

APPENDIX A: DESIGNATION OF CLASS NUMBERS FOR EACH CLASS LABELS

Table A.1 depicts assigned class numbers for each class label. The class numbers can be considered as an identifier for the class labels. In the *data.yaml*, there is an array variable called *names* which is populated with class labels. YOLOv5 algorithm adapts the index which indicates the position of the each class label within the *names* as a class number.

Table A.1. Designated class numbers for each class label

Class Number	Class Label
0	Background Image
1	Checked View
2	Icon
3	Input Field
4	Image
5	Text
6	Text Button
7	Sliding Menu
8	Page Indicator
9	Upper Task Bar
10	Pop-Up Window
11	Switch

APPENDIX B: TEXT RECOGNITION IN GUI ELEMENTS

Algorithm B.1 is written to extract text in GUI elements. It employs OpenCV and Pytesseract libraries and consists of *detectText* function. Loop through each file path of cropped GUI images are processed and recognized text values are saved in a text file.

Algorithm B.1. Python script to detect text in GUI elements

```

1 def detectText(imagePath):
2     # assigning an image from the image path
3     img = cv2.imread(imagePath)
4
5     # grayscale image
6     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
7
8     # threshold image
9     ret, thresh1 = cv2.threshold(gray, 0, 255,
10    cv2.THRESH_OTSU | cv2.THRESH_BINARY_INV)
11
12    # structuring element method
13    rect_kernel = cv2.getStructuringElement(cv2.MORPH_RECT
14    , (12, 12))
15
16    # find contours in image
17    contours, hierarchy = cv2.findContours(thresh1,
18    cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
19
20    img2 = img.copy()
21
22    for cnt in contours:
23        x, y, w, h = cv2.boundingRect(cnt)
24
25        # Draw the bounding box on the text area
26        rect=cv2.rectangle(img2, (x, y), (x + w, y + h)
27        , (0, 0, 255), 3)
28
29        # Crop the bounding box area
30        cropped = img2[y:y + h, x:x + w]
31
32        file = open("output.txt", "a")
33
34        # Using tesseract on the cropped image area to get text
35        text = pytesseract.image_to_string(cropped)
36
37        # Adding the text to the file
38        file.write(text)
39        file.write("\n")
40        file.close

```

APPENDIX C: GRABBING THE COLOR PALETTE FROM THE GUI COMPONENTS

Algorithm C.1 is developed to grab dominant colors in the GUI elements. It adapts *colorthief* library and consists of *detectColor* function. Loop through each file path of cropped GUI images are fed into *detectColor* method and the RGB values of the first two dominant colors are saved in a text file.

Algorithm C.1. Python script to grab dominant colors in the GUI elements

```
1 from colorthief import ColorThief
2
3 def detectColor(imagePath)
4
5     # assigning an image from the image path
6     color_thief = ColorThief(imagePath)
7
8     # grabbing the first two dominant colors from the image
9     palette = color_thief.get_palette(color_count=2, quality=1)
10
11     # convert list of tuples to string
12     colorStrings = str(palette).strip('[]')
13
14     # open the text file
15     file = open("color.txt", "a")
16
17     # Adding the text to the file
18     file.write(colorStrings)
19     file.write("\n")
20
21     # closing the file
22     file.close
```


APPENDIX D: GENERATION OF UI LAYOUT FILES FOR ANDROID AND IOS PLATFORMS

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
3   xmlns:app="http://schemas.android.com/apk/res-auto"
4   xmlns:tools="http://schemas.android.com/tools"
5   android:layout_width="1080dp"
6   android:layout_height="1794dp"
7   tools:context=".MainActivity">
8
9   <ImageView
10     android:id="@+id/imageView"
11     android:layout_width="412dp"
12     android:layout_height="214dp"
13     app:srcCompat="@drawable/kik"
14     tools:layout_editor_absoluteX="350dp"
15     tools:layout_editor_absoluteY="503dp" />
16
17   <Button
18     android:id="@+id/button2"
19     android:layout_width="1062dp"
20     android:layout_height="122dp"
21     android:backgroundTint="#25BCFA"
22     android:text="@string/sign_up"
23     android:textColor="#E5F8F8"
24     android:textColorLink="#E5F8F8"
25     android:textSize="40sp"
26     tools:layout_editor_absoluteX="7dp"
27     tools:layout_editor_absoluteY="1061dp" />
28
29   <Button
30     android:id="@+id/button3"
31     android:layout_width="947dp"
32     android:layout_height="122dp"
33     android:backgroundTint="#EEFBF9"
34     android:text="login"
35     android:textColor="#43B9E2"
36     android:textColorLink="#43B9E2"
37     android:textSize="40sp"
38     tools:layout_editor_absoluteX="63dp"
39     tools:layout_editor_absoluteY="1210dp" />
40
41   <EditText
42     android:id="@+id/editTextTextPersonName"
43     android:layout_width="446dp"
44     android:layout_height="wrap_content"
45     android:ems="10"
46     android:gravity="center"
47     android:inputType="textPersonName"
48     android:text="@string/chat_with_friends"
49     android:textAlignment="center"
50     android:textColor="#42434C"
51     android:textSize="52dp"
52     tools:layout_editor_absoluteX="322dp"
53     tools:layout_editor_absoluteY="772dp" />
54
55   <EditText
56     android:id="@+id/editTextTextPersonName2"
57     android:layout_width="444dp"
58     android:layout_height="wrap_content"
59     android:ems="10"
60     android:inputType="textPersonName"
61     android:text="@string/made_with_love_in_waterloo"
62     android:textAlignment="center"
63     android:textColor="#C4C3CB"
64     android:textSize="33dp"
65     tools:layout_editor_absoluteX="318dp"
66     tools:layout_editor_absoluteY="1710dp" />
67 </androidx.constraintlayout.widget.ConstraintLayout>

```

Figure D.1. Generation of Android UI layout file for the image in Figure 5.10a

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <document type="com.apple.InterfaceBuilder3.CocoaTouch.Storyboard.XIB" version="3.0" toolsVersion="20033" targetRuntime="iOS.CocoaTouch" propertyAccessControl="none" useAutolayout="YES" useTraitCollections="YES" useSafeAreas="YES" colorMatched="YES" initialViewController="BYZ-38-10r">
3      <device id="retina4_1" orientation="portrait" appearance="light"/>
4      <dependencies>
5          <plugin identifier="com.apple.InterfaceBuilder3.CocoaTouchPlugin" version="20020"/>
6          <capability name="safe area layout guides" minToolVersion="9.0"/>
7          <capability name="System colars in document resources" minToolVersion="11.0"/>
8          <capability name="Documents saved in the Cloud & Format" minToolVersion="9.0"/>
9      </dependencies>
10     <scenes>
11         <!--View Controller-->
12         <scene sceneID="tne-Q1-fo">
13             <objects>
14                 <viewController id="BYZ-38-10r" customClass="ViewController" customModule="StoryboardTesting" customModuleProvider="target" sceneMemberID="viewController">
15                     <view key="view" contentMode="scaleToFill" id="86C-W-cv">
16                         <rect key="frame" x="0.0" y="0.0" width="388" height="1794"/>
17                         <autoresizingMask key="autoresizingMask" widthSizable="YES" heightSizable="YES"/>
18                         <subviews>
19                             <UIImageView key="imageView" userInteractionEnabled="NO" contentMode="scaleAspectFit" horizontalHuggingPriority="251" verticalHuggingPriority="251" fixedFrame="YES" image="3728" translatesAutoresizingMaskIntoConstraints="NO" id="81a-ss-26a">
20                                 <rect key="frame" x="108" y="583" width="112" height="214"/>
21                                 <autoresizingMask key="autoresizingMask" flexibleMaxX="YES" flexibleMaxY="YES"/>
22                             <imageView key="imageView" userInteractionEnabled="NO" contentMode="left" horizontalHuggingPriority="251" verticalHuggingPriority="251" fixedFrame="YES" text="Chat with friends" textAlignment="center" lineBreakMode="tailTruncation" baselineAdjustment="alignBaselines">
23                                 <adjustSizeToFitText="NO" translatesAutoresizingMaskIntoConstraints="NO" id="ub-42-cia">
24                                     <rect key="frame" x="121" y="773" width="144" height="52"/>
25                                     <autoresizingMask key="autoresizingMask" flexibleMaxX="YES" flexibleMaxY="YES"/>
26                                     <fontDescription key="fontDescription" type="system" pointSize="12"/>
27                                     <color key="textColor" red="0.108213924117446" green="0.282748988392137" blue="0.29883921568627481" alpha="1" colorSpace="calibratedRGB"/>
28                                     <nil key="highlightColor"/>
29                                 </imageView>
30                             <imageView key="imageView" userInteractionEnabled="NO" contentMode="left" horizontalHuggingPriority="251" verticalHuggingPriority="251" fixedFrame="YES" text="Made with love in Waterloo" textAlignment="center" lineBreakMode="tailTruncation" baselineAdjustment="alignBaselines">
31                                 <adjustSizeToFitText="NO" translatesAutoresizingMaskIntoConstraints="NO" id="wef-a1-0j">
32                                     <rect key="frame" x="118" y="1718" width="144" height="33"/>
33                                     <autoresizingMask key="autoresizingMask" flexibleMaxX="YES" flexibleMaxY="YES"/>
34                                     <fontDescription key="fontDescription" type="system" pointSize="12"/>
35                                     <color key="textColor" red="0.108213924117446" green="0.282748988392137" blue="0.29883921568627481" alpha="1" colorSpace="calibratedRGB"/>
36                                     <nil key="highlightColor"/>
37                                 </imageView>
38                             <button key="button" contentMode="scaleToFill" fixedFrame="YES" contentHorizontalAlignment="center" contentVerticalAlignment="center" buttonType="system" lineBreakMode="middleTruncation" translatesAutoresizingMaskIntoConstraints="NO" id="Q08-M6-mLi">
39                                 <autoresizingMask key="autoresizingMask" flexibleMaxX="YES" flexibleMaxY="YES"/>
40                                 <color key="backgroundColor" red="0.933333333333333" green="0.7643137254807" blue="0.976476582352948" alpha="1" colorSpace="calibratedRGB"/>
41                                 <state key="normal" title="Button"/>
42                                 <buttonConfiguration key="configuration" style="plain" title="LOG IN">
43                                     <fontDescription key="titleLabelDescription" type="system" pointSize="18"/>
44                                     <color key="baseForegroundColor" red="0.262748988392137" green="0.725490196078431" blue="0.886274898839213" alpha="1" colorSpace="calibratedRGB"/>
45                                 </buttonConfiguration>
46                             <button key="button" contentMode="scaleToFill" fixedFrame="YES" contentHorizontalAlignment="center" contentVerticalAlignment="center" buttonType="system" lineBreakMode="middleTruncation" translatesAutoresizingMaskIntoConstraints="NO" id="P9B-co-uP7">
47                                 <autoresizingMask key="autoresizingMask" flexibleMaxX="YES" flexibleMaxY="YES"/>
48                                 <color key="backgroundColor" red="0.148988392156862" green="0.737254901960784" blue="0.988392156862748" alpha="1" colorSpace="calibratedRGB"/>
49                                 <state key="normal" title="Button"/>
50                                 <buttonConfiguration key="configuration" style="plain" title="SIGN UP">
51                                     <fontDescription key="titleLabelDescription" type="system" pointSize="18"/>
52                                     <color key="baseForegroundColor" red="0.898392156862748" green="0.972549019607843" blue="0.972549019607843" alpha="1" colorSpace="calibratedRGB"/>
53                                 </buttonConfiguration>
54                             <button>
55                                 <fontDescription>
56                                 <viewLayoutGuide key="safeArea" id="47A-OC-88P"/>
57                                 <color key="backgroundColor" systemColor="systemBackgroundColor"/>
58                             </view>
59                             <view key="imageView" width="388" height="1794"/>
60                         </subviews>
61                         <viewController>
62                             <viewControllerIdentifier="BYZ-38-10r" sceneMemberID="firstResponder"/>
63                         </objects>
64                         <point key="canvasLocation" x="18" y="78"/>
65                     </scene>
66                 </viewController>
67             </scenes>
68             <resources>
69                 <image name="3728" width="112" height="214"/>
70                 <systemColor name="systemBackgroundColor"/>
71                 <color name="1" alpha="1" colorSpace="custom" customColorSpace="genericGamma22GrayColorSpace"/>
72             </resources>
73     </document>

```

Figure D.2. Generation of iOS UI layout file for the image in Figure 5.10a