UI2CODE: Computer Vision Based Reverse Engineering of User Interface Design

Mulong Xie

A thesis submitted for the Course
COMP4540 Software Engineering Research Project
The Australian National University

October 2019

Except where otherwise indicated, this thesis is my own original work.

Mulong Xie
26 October 2019

Acknowledgments

Great gratitude to my parents and friends: you are my cherished backings who have persistently motivated and stimulated me in tough times.

I am grateful to Dr Chunyang Chen for his guidance and suggestions that supported me to conduct this work at the beginning.

Much appreciation to my supervisor Dr. Zhenchang Xing, who helped me to initiate my research experience in the Australian National University and have constantly provided me instructive guides and inspiring thoughts. I am glad to handle this individual project under your supervision and have gained a lot valuable knowledge and enlightenment which I believe will profoundly benefit my future.

Abstract

Human-computer interface, functioning as the channel for communication and interaction between people and computer system, has been increasingly applying the graphical user interface (GUI). However, there is a significant gap between the GUI design and implementation. In particular, developers must manually convert GUI design drawings into working code and laboriously revise the implementation whenever the designs undergo some (even trivial) changes. Most existing GUI development tools focus attention on providing fancy design images and convenient drag and drop editor, but they do not directly offer the maintainable and usable source code, which is not suitable for the real industrial GUI development project. In this thesis, we propose a novel computer vision based GUI reverse engineering system, UI2CODE, to bridge the gap between GUI design and implementation, and relieve some pains of manual programming and code maintenance. UI2CODE is not built on end-to-end deep learning object detection approaches, because our study shows that these deep learning object detection methods do not work well on GUI design images due to these pictures' artificial nature and particular characteristics. Instead, our approach utilizes conventional computer vision and image processing algorithms which can bypass the essential problems with the machine learning techniques and accurately detect GUI elements in complex GUI designs. Experiments on various datasets, including Web UIs, mobile app UIs, and artistic UI design drawings, prove the effectiveness of UI2CODE in terms of accuracy of GUI component detection and layout reconstruction. UI2CODE enables a more efficient development workflow where GUI design images can be automatically converted to maintainable code within a short time-frame, which is impossible before.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Statement

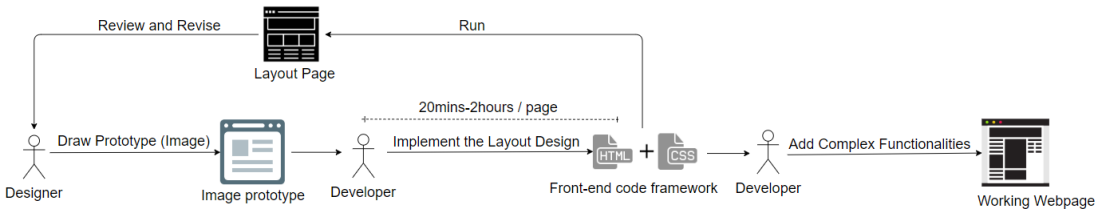
UI2CODE is a computer vision based reverse engineering technique that generates the front-end code from human-computer interface images.

1.2 Introduction

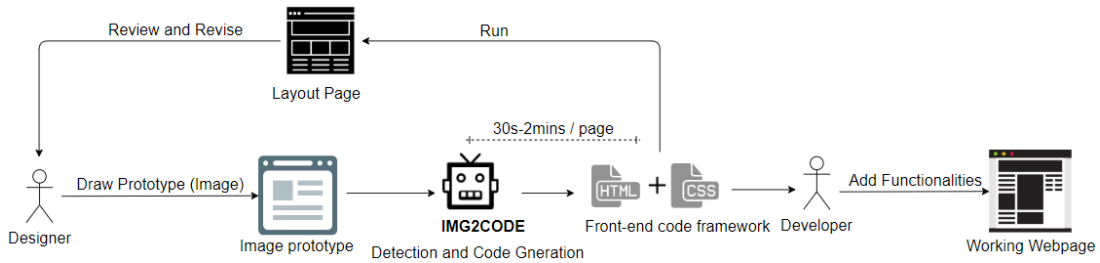
Human-computer interface is the means of communication between people and low-level system through which the system conveys information to the user and receives human instructions [Laurel and Mountford, 1990; Brown, 1999]. As it has evolved with the development of information technology, the manner of providing access for human to interact with computer has increasingly transformed into graphical interface. For instance, the graphical user interface (GUI) displays abundant contents and offers readily approaches for people to use sophisticated functionalities easily [Farrugia, 2007]. Such advancement brings convenience to the user while changes the way how the user interface (UI) designer and developer work. This thesis hence introduces UI2CODE, a novel technique that eases the UI development by leveraging computer vision techniques and code generation approaches.

The common workflow of user interface development involves roles designing the layout and style of the GUI, as well as coders responsible for implementing this design into program and building the connection between the front-end presentation and the back-end data and functionalities [Selene M., 2018; Gordiyenko, 2019a], as illustrated in Figure 1.1. UI2CODE is expected to speed up this process. This system comprises two high-level steps: (1) automatically localizing the positions and recognizing the semantic meanings of GUI components in the input image. The image can either be a screenshot of an existing user interface or a prototype drawn by a drawing application, such as Photoshop. (2) Generating the usable and maintainable front-end code, such as HTML and CSS, in accordance with the detection results. Figure 1.2 presents the visualized process of this system.

UI2CODE significantly shortens the time of the process that implements the code of a given UI design, which takes several seconds to few minutes to handle the input image in any size, while the same work performed by human spends a much



(a) Conventional process of web development



(b) Web development with the assistance of UI2CODE

Figure 1.1: The illustration that presents how UI2CODE facilitates web development. Figure 1.1(a) is the typical industrial development process in which the developer takes pains to implement the design drawing of a user interface and continuously revise it until reaching the final layout solution with the designer. Figure 1.1(b) shows the UI2CODE would replace the role of the developer when implementing the layout design, which dramatically shortens the time and helps the developer get rid of trivial work and eventually boosts the efficiency of the development process.

longer time. In addition, the generated code is highly readable and maintainable, and it is also created to be easy to extend so that the developer can conduct further functionality development on the basis of the code framework. Plenty of pains can be relieved by applying this system, and the developer can get rid of some repetitive and rudimentary work.

A variety of UI development assistant tools are available now, such as Wix, WordPress and JIMDO [GmbH, 2019], but most of them aim to provide fancy pre-defined templates and convenient drag-and-drop editors to people rather than offering the high-quality source code. They are suitable for those who simply desire for making a nice-looking website but do not claim the ownership of the source code. However, the industrial UI development project requires the full knowledge of the program for the sake of maintenance and future extension. Therefore, UI2CODE attempts to fill in the gaps between the designing phase and the programming phase of professional UI development.

This work includes several major contributions:

1. A computer vision based user interface code generator consisting of two modules: the UI components detector that accurately detects and classifies the graphical interface components in the image, and the code generator that creates the corresponding front-end program achieving the identical visual effect

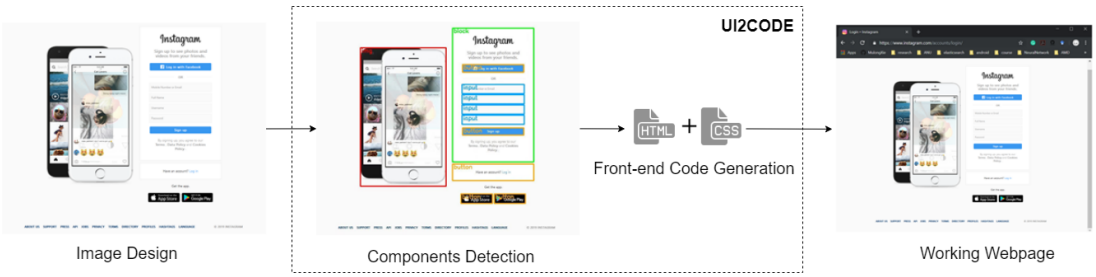


Figure 1.2: The visualized workflow of UI2CODE. This system takes the user interface design image as input, and the pipeline proceeds from detecting the UI components in the input image and then generates the corresponding front-end code, i.e. HTML and CSS, which can be run in a browser to see the working web page with the same visual effect as the design.

of the input UI.

2. Several brand new datasets containing various web and mobile user interface images, including screenshots of web pages, real UIs of mobile applications and artistic UI design drawings.

3. Analyses of some popular deep learning based object detection approaches, pointing out their inadequacies while directly applied in this task and stating the underlying reasons.

4. An effective UI component detection technique comprising multiple novel image processing algorithms and a machine learning classifier, fitting into the particular characters of graphical user interface, reaching a high accuracy and recall that significantly surpass the performance of existing object detection models.

5. An efficient HTML code generator that produces the maintainable and easy-to-extend UI interface program framework, including an approach of dividing the entire image to hierarchical layout blocks, and an HTML/CSS code generator based on DOM tree.

To this end, this project is composed of two major parts, the UI components detection part and the code generation part.

1.2.1 UI Components Detection

As the core part of this system, the UI components detection module performs the semantic segmentation of the input image. However, detecting the graphical interface elements is challenging and relatively fresh where few works have been done in this field, and similar works are all based on end-to-end object detection methods [White et al., 2019]. Furthermore, several particular properties of the graphical

user interface hamper the effectiveness of existing related techniques, in particular, the deep learning based object detection approaches, such as [Ren et al., 2015; Redmon et al., 2016; Liu et al., 2016]. Therefore, a domain-specific technique is required to handle the particular challenges in this task and fit the unique requirements of human-computer interface design.

This thesis presents a summary of the graphical user interface's characters. These attributes not only exhibit the distinctive visual effect of the UI components but also represent the special demands for the detection results. For instance, the artificial interface elements must be absolutely precise in terms of the size and the spacial position because they are created by the program, which asks for the detection and localization to be strictly accurate.

Prior to my approach, this thesis first analyzes the popular related detection methods, especially those based on deep learning, and attempts to identify the flaws that prevent them from being adopted in this case. For example, I had an observation revealing some interesting issues of bounding box regression which is essentially a statistical evaluation and can hardly predict the flawless boundary of components. Thus, I proposed a technique utilizing conventional computer vision and image processing methods to bypass the defects of the approaches above.

I built a UI components detection system consists of two independent branches: graphical components detection and text recognition. The overall structure is shown in Figure 5.1. The graphical components detection is responsible for localizing and classifying the graphical elements in the input image. Rather than directly using end-to-end machine learning approach, this branch is a pipeline containing three sections: pre-processing, component detection and classification. The first two parts utilize image processing algorithms to segment the components and select the UI element candidates accurately, and the classification part constructs a convolutional neural network to categorize those candidates into the defined classes for code generation.

On the other hand, the text recognition branch is independent of the graphical component detection for the sake of precision and efficiency. I apply a powerful natural scenes text detection approach, Connectionist Text Proposal Network (CTPN) [Tian et al., 2016] to perform this work. In the end, the system merges the results from both branches and conducts the cross-check to polish the final detection result.

1.2.2 Code Generation

This section functions as the layout reconstruct layer as well as the output layer that produces the deliverable code. So far, I only conducted experiments on web UIs in HTML and CSS. Thus, this part assembles all the detection and segmentation results to manufacture the front-end program for a web page. As the UI components detection, there are few directly similar works can be used as references, but fortunately, I can draw inspiration from related topics. Works involving the front-end program are most about information gathering and extraction. For example, plenty of works aim to efficiently parse the HTML files of a web page [Gupta et al., 2003; Cosulschi et al., 2006], as well as abundant works focus attention on collecting information

from world wide web (WWW) by improving the crawling agent [Olston and Najork, 2010; D'Haen et al., 2016]. I referred to their thoughts and treated the HTML file as a Document Object Model (DOM) tree [Whitmer, 2009] to generate the professional code in a systematically way.

The HTML DOM constructs a tree structure to describe a web page in which each branch node represents an element, and the leaf node shows the content and attributes of its parent element [Lee, 2012]. Figure 6.6 illustrates a typical DOM tree for a working web page. It is a standard way to express a web UI, as well as an efficient data structure to reproduce the source code. Therefore, I proposed a pipeline consisting of three steps: (1) divide the entire UI into hierarchical layout block according to cutting lines; (2) construct an HTML DOM tree based on the hierarchy; (3) convert the tree into HTML and CSS program.

I created multiple algorithms to acquire the block division with hierarchy. First, an approach was adopted to detect the cutting line from the pre-processed binary map of the input image. There are some popular image processing algorithms to detect lines in a nature scene, such as Hough transform [Duda and Hart, 1972], but they are rather complicated and unnecessary in our case. I observed that the cutting lines in a UI are always borders of elements, which are either horizontal or vertical. So I constructed a specific and efficient method to detect these lines. Second, the lines are combined with gaps among elements to divide the entire image into multiple blocks, and I applied some tricks to establish the hierarchy of blocks on the basis of their spatial positions.

Results from the previous UI components detection pipeline are incorporated into the layout segmentation to construct the aforementioned HTML DOM tree. Elements' positions and sizes are transformed into relative values in their parent blocks to be consistent with the usual practices of web development. Furthermore, in order to enhance the maintainability and expansibility of the manufactured code, the system assigns a unique ID to each element and implements the interface for some functional widget, such as input box and button, for further functionality development.

Serially passing through the above UI components detection branch and code generation branch, the input UI image is converted into a set of high-quality front-end code to reproduce the identical visual effect and expected functionality.

1.3 Thesis Outline

This thesis includes totally seven chapters to detail the thought, datasets, methodologies and experimental results.

- **Chapter 2 Background and Related Work** states the motivation of this work that mainly aims to facilitate professional UI development. Then it roughly mentions some related work in all involved fields, such as graphical interface

image datasets and object detection methods, but details of these works will be elaborated in each corresponding chapter.

- **Chapter 3 Difference from Object Detection** summarizes four particular properties of the graphic user interface and introduces several state-of-the-art deep learning based object detection approaches in the way that identifies their drawbacks hindering them from performing effective detection as they do in natural scenes. Thus to inspire my own work built on alternative methods to settle down of bypass these issues.

- **Chapter 4 Data Collection** lists several human-computer interface image datasets, including a self-built web UI dataset containing various real screenshots crawled from world wide web, a published large mobile application dataset Rico and a collected artistic UI design drawing dataset from Dribbble. This chapter also shows some encountered problems in the web data collection process and some measures applied to overcome these issues.

- **Chapter 5 User Interface Components Detection** presents the technical details of UI2CODE's UI components detection section, a pipeline composed of two branches to achieve the precise localization and classification of graphic elements and text recognition. This chapter elaborates the architecture of the detection part based on multiple novel and specific image processing algorithms and a machine learning classifier, as well as the mechanism and usage of a text recognition technique CTPN. In the end, an approach is presented to incorporate the results from both branches to produce the final result.

- **Chapter 6 Code Generation** introduces the second part of UI2CODE where the input UI image's layout is reproduced, and all the previous results are assembled to generate the deliverable front-end code. It covers two sections, the first one presents multiple algorithms that hierarchically segment the entire image into several layout blocks according to cutting lines and gaps among elements; the second section describes the process that constructs an HTML DOM tree and converts this tree into working HTML/CSS code. This part of the work is still immature and requires future research to solve the problems mentioned in the limitation section.

All code of this project can be found in GitHub repository: https://github.com/MulongXie/Research-ReverselyGeneratingWebCode.

Chapter 2

Background and Related Work

Before presenting the work's progress and findings, the thesis provides this chapter for introducing the motivation of the entire project and several state-of-the-art works in all corresponding topics. However, this chapter only states brief of these related works, their details and analyses will be elaborated in relevant chapters.

2.1 Motivation

The fundamental motivation of this work is to propose a system to help the developer get rid of trivial and repetitive work, whereby facilitating the user interface development process. As mentioned in Chapter 1, the current workflow of the UI development, especially for industrial-level projects which require high-quality code with the efficient development process, have potential to be improved in terms of developing time and product maintainability by assistant tools. However, the existing graphical web builders, such as various Wix and Wordpress, mostly focus on designing phase rather than generating the source code. Besides, although some modern IDEs, such as Eclipse, Xcode and Android Studio have powerful interactive builders for GUI, they are relatively complex to use and cannot directly fill in the gap between designers and developers [Nguyen and Csallner, 2015; Zeidler et al., 2013]. Therefore, we created the UI2CODE to fill in the gap between designing and programming of UI development.

The standard UI development process always comprises seven steps: first gathering requirements from the client; creating the mock-up and selecting technology stack; designing the layout; filling contents of this website; implementing the working code of the UI; testing and reviewing with the client; finally daily maintenance [Gordiyenko, 2019b]. UI2CODE aims to facilitate the coding, testing and maintenance phases in the way that automatically manufactures the code from design drawing, and the generated code is expected to be convenient to maintain and expand to assistant the testing and daily maintenance as illustrated in Figure 2.1.

UI2CODE is a reverse engineering technique taking the human-computer interface image as input, recognizing UI components and generating the usable front-end code. It is expected to ease the UI development and facilitate the developer or designer to implement the design into working code, as illustrated in 1.1. Because

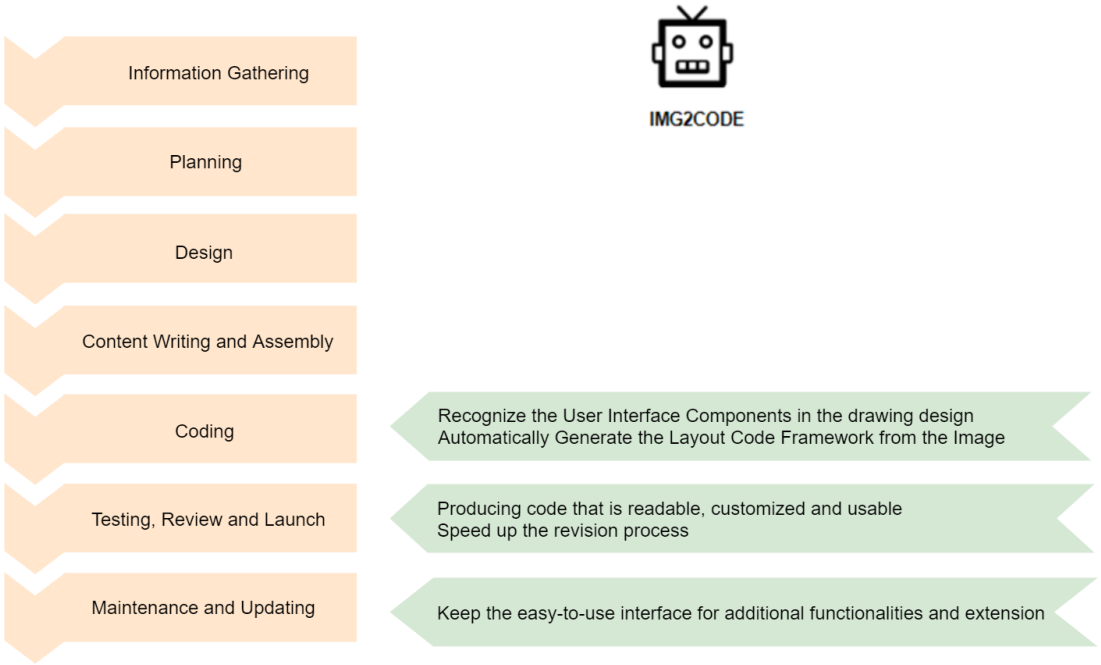


Figure 2.1: Illustration of how the UI2CODE facilitates the modern professional UI development process. The system focus attentions on facilitating the code implementation and testing phase, as well as maintenance phase of a project, while plenty of existing web builders can only be used in the design stage.

this system would work under the industrial environment and used by specialized people, the quality of its outcome ought to reach the professional level. To be more specific, the identification of components on the design drawing should be as accurate as possible, and the generated code should be as maintainable as possible.

To this end, I designed the UI2CODE as a pipeline composed of two major steps, the UI components detection phase and the code generation phase. I leveraged a variety of related techniques and proposed various novel methods to address the problems encountered in each section of this work. Eventually, an intelligent graphical human-computer interface development assistance was built to fulfil the needs of developers and designers.

2.2 Related work

This work covers various topics, including GUI datasets, image processing, object detection and reverse engineering. In this process, a variety of related works in each field are investigated and referred. This section only lists and briefs papers which are mentioned in the subsequent text, and their details and insights are elaborated in relevant chapters later.

2.2.1 UI Reverse Engineering

Reverse engineering for the pixel-based user interface is relatively fresh. The most similar work with UI2CODE is REMAUI [Nguyen and Csallner, 2015], a system aiming to perform reverse engineering on mobile application user interfaces. It was the first technique achieving so. This work leveraged computer vision methods to locate and extra UI components as a hierarchy of nested bounding boxes, and utilized optical character recognition (OCR) to detect text regions. The high-level workflow is similar to the UI2CODE, but there are some significant limitations in this work. First, the REMAUI is not capable of classifying the extracted UI elements; instead, it focuses on reproducing the layout of input design. Second, the performance of this method, in terms of accuracy of UI component extraction, is sceptical. This work only adopted some primitive image processing algorithms, such as Canny edge detection [Canny, 1986] without any further improvement, which is inadequate and problematic to handle complicated designs, especially for UIs with image elements. I addressed this problem and detail the process in section 5.2.

In addition, some UI reverse engineering techniques depend on predefined models of UI elements. Prefab [Dixon and Fogarty, 2010] is a representative of such methods. However, their drawbacks are apparent, because the graphic designs can be diverse and creative, and a tool based on some predefined templates is too limited. Besides, the predefined templates are usually hard to expand, which does not fit the requirement of industrial level UI development [Laurel and Mountford, 1990; Brown, 1999].

Another branch of UI reverse engineering is aiming paper-and-pencil design drawings [Landay and Myers, 2001; Coyette et al., 2007]. Although design with assistant software, such as PhotoShop, has been increasingly popular and prevailed, the paper-based sketches are still be adopted in many companies, because of habits and background of some graphic designers [Landay and Myers, 1994; Campos and Nunes, 2007]. Works in this topic infer UI code from hand-drawings. But some limitations exist in the paper-based method. First, it is not as convenient to change a drawing on paper as revising a design in software. Second, designers can only present the overall layout of UIs rather than detailed and complicated illustrations on paper. For example, designers can directly import some images into the design by using digital applications, but they can hardly do so on paper. Therefore, UI2CODE focuses on facilitating UI development based on digital design, which is more complex and universal.

2.2.2 Data Collection

Data-driven models of design provide UI designers with access to relevant examples and hence helps them understand best practices and trends. These models also enable systems that foresee the effectiveness and attractiveness of creations on the basis of observation on past designs [R. and P., 2014]. To this end, researchers have been building various datasets to expose the UI designs at scale [Deka et al., 2017].

This work involved both web UIs and mobile UIs, which requires related datasets in both types of data to sustain analyses and experiments.

Researchers in relevant fields have published a variety of mobile datasets collecting data for different purposes. For example, some repositories gathered Google Play metadata (e.g. reviews, ratings) [Fu et al., 2013] and some works built datasets of design data [Sahami Shirazi et al., 2013; Alharbi and Yeh, 2015]. UI2CODE only focuses on UI designs in image format, so I mainly leveraged a currently largest mobile UI repository named Rico [Deka et al., 2017], where including 72k unique UI images. The strategy applied to build such large dataset combined human exploration and automated exploration.

Besides, I collected a set of real web page screenshots by a web crawler to investigate the character of web UIs and to evaluate the effectiveness of UI2CODE, as well as a dataset of artistic design drawings from Dribbble [Dribbble, 2019] to explore other potential use cases. Some works about web crawling are referred to in this process [Olston and Najork, 2010; D'Haen et al., 2016] to parse the downloaded web page file and extract the information more efficiently. Those works mainly treat the source HTML files as DOM tree [Whitmer, 2009], and all the elements and their attributes are stored as nodes. I followed this methodology to build my crawling agent and analyze the HTML file to retrieve the target information, such as the position and size of the UI element.

2.2.3 UI Components Detection

UI components detection is also a relatively fresh topic, one work attempting to achieve so is an Image-Based Widget Detector [White et al., 2019]. It also takes the GUI image as input and tries to detect the UI widgets. This work adopted YOLO [Redmon et al., 2016], a deep learning based object detection approach, to achieve its purpose. However, as I spent an entire Chapter 3 analyzing, the object detection methods based on machine learning are not suitable for graphical user interfaces. UI has several specific characters and requirements, such as heterogeneous presences and strict demanding for precision, which the statistical or estimated machine learning techniques cannot satisfy.

Other indirectly related works are object detection approaches, they are typically divided into two categories: region-based methods represented by RCNN, Fast RCNN and Faster RCNN [Girshick et al., 2014a; Girshick, 2015; Ren et al., 2015] and one-shot methods such as YOLO and SSD [Redmon et al., 2016; Redmon and Farhadi, 2018; Liu et al., 2016]. But the localization of objects in these models is less or more dependent on the bounding box regression mechanism [Lee et al., 2019], which can not predict the absolutely accurate boundaries of components. So I proposed a technique combining the computer vision localization algorithms with the machine learning classifier to bypass the problem. I will detail this in chapter 3 thoroughly.

In detail, the UI components detection pipeline consists of several steps, as shown in Figure 5.1. Each step involves different topics and refers to various related works.

For instance, the pre-processing parts produce a binary map according to the original image' s gradients map for the purpose of distinguishing the foreground from the background. Usually, Canny [Canny, 1986] edge detector is used to identify individual objects by calculating their margin gradients, but the result of this method is too fine-grained, and it includes too many texture details that negatively affect the UI components segmentation. Figure 5.3 shows the comparison between my method and other popular counterparts.

Besides, in the graphical component detection part, I proposed several novel image process algorithms which are more efficient in this task compared to related works. For instance, the popular practice to recognize a rectangle is applying the Hough transform [Duda and Hart, 1972] or the approxPolyDP function in OpenCV [Dev, 2014] based on Douglas-Peucker algorithm [Douglas and Peucker, 1973]. They all involve too many computations to fit the universal situations in the real world, but our case only requires to estimate whether a component is rectangular or not. Thus I constructed a simple and specific rectangle recognition technique to fulfil the need more efficiently.

In addition, to classify the components, I built a classifier utilizing machine learning. Several approaches were tried, including HOG feature [McConnell, 1986; Freeman and Roth, 1994] or SIFT feature [Lowe, 2004, 1999] combined with SVM Patel [2017], as well as a simple four-layer convolutional neural network (CNN). The experiments prove the CNN can better handle this task while the others are more suitable for natural scenes.

2.2.4 Code Generation

This part comprises two sections, layout segmentation and web code generation. In order to divide the entire image into hierarchical layout blocks, the segmentation section addressed several concerns by utilizing multiple approaches. First, we need to detect cutting lines on a UI, one way to achieve so is Hough Transform [Duda and Hart, 1972]. Hogh transform project the parameters of a line onto a special coordinate system and iterate each foreground point to search lines. However, it is rather computationally expensive and unnecessary for our task, because cutting lines on a UI are always elements' borders which are either horizontal or vertical. Therefore I proposed an efficient algorithm only detect these two types of lines.

Regarding web code generation, works involving front-end source code are most focusing attentions on either collecting Olston and Najork [2010]; D'Haen et al. [2016] or parsing UIs [Cosulschi et al., 2006; Gupta et al., 2003] while few directly generate code by applying automated agent. But the aforementioned works inspired me to organize the HTML code into a DOM tree Whitmer [2009]. Furthermore, some web development guides [Musciano and Kennedy, 2007; W3Cschool, 2019] provided me with the standards to improve the quality of the generated code. For example, using the relative values rather than absolute values to arrange the UI layout, and assigning a unique ID if necessary to the element to render and control the web page in a more systematical way.

2.3 Summary

This chapter introduces the motivation of UI2CODE in real UI development. The system is expected to be an assistant tool to fill in gaps between the conceptual design image and the working code, which significantly shortens the development timeframe and relieves lots of pains of designers and coders. Regarding related work, reverse engineering for UI is a fresh topic, and the existing similar works are instructive but not robust. Moreover, a variety of topics are involved in this work, such as UI dataset, components detection and web code generation, so I referred to relevant works in each section to gain inspirations and draw comparisons. These works are also elaborated in their corresponding chapters.

Chapter 3

Difference from Object Detection

Regarding detection, the first idea coming to mind for most contemporary researchers in related fields is the set of deep learning based approaches. Recent thriving and development of neural network endow significant capability to object detection models utilizing deep learning [Gandhi, 2018]. Popular techniques, represented by the RCNN family (RCNN, Fast RCNN, Faster RCNN etc.) [Girshick et al., 2014a; Girshick, 2015; Ren et al., 2015] and the one-step end-to-end methods (YOLO, SSD etc.) [Redmon et al., 2016; Liu et al., 2016] as well as the semantic segmentation methods (Mask RCNN etc.) [He et al., 2017], have performed remarkable ability to capture objects under a variety of complex environments. However, on the basis of my experiments and analyses, most of those approaches are designed to detect targets in the natural scenes, but they can hardly fit directly into the detection task of the artificial graphical human-computer interface, such as the user interface (UI) components detection. Multiple factors contribute to such unsatisfied performance in this mission, including the specified requirements and practices when a user interface is designed, and the essential mechanisms of deep learning based object detection that is inappropriate to be used in this case.

This chapter summarizes these particular characters in the human-computer interface and briefly introduces the principles and problems of some state-of-the-art deep learning based object detection methods when applied in this case. Then the conclusion about the necessity of the domain-specific approach is drawn at the end on the basis of analyses and comparisons.

3.1 Characters of the Human-computer Interface

Based on the statistics, we observe that the artificial interface have some common visual attributes that differ from the natural scenes. These distinctions make the conventional deep learning approaches hard to perform their abilities as effectively as they do in natural images.

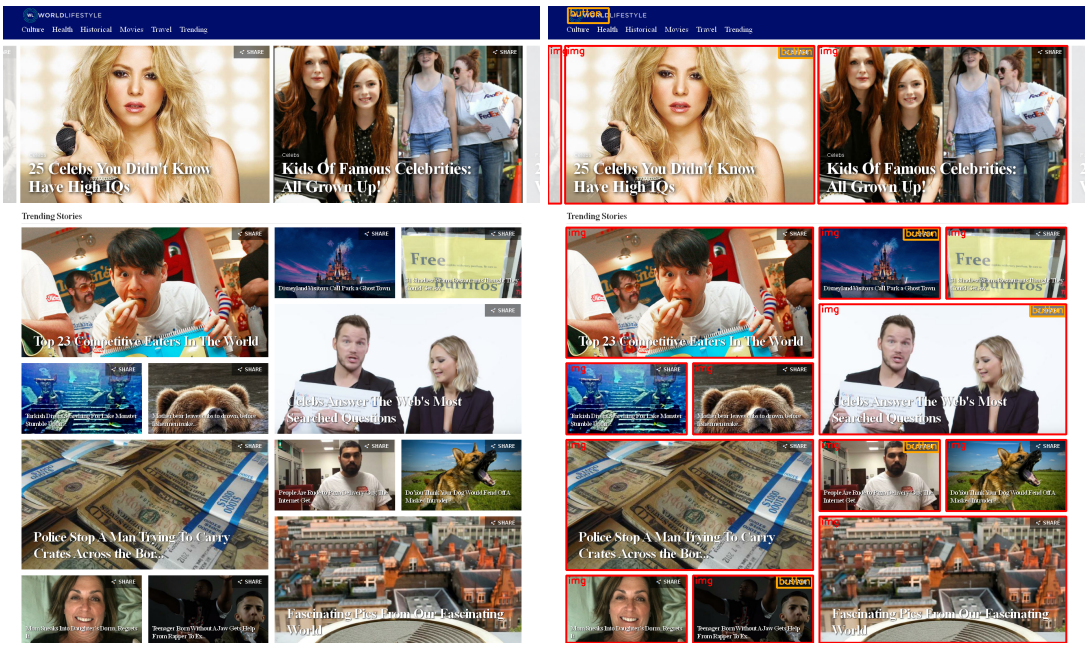
Property 1: The contents of the graphical user interface are heterogeneous. To be more specific, interface design could contain various components, including the functional elements, such as the button and input box, and the static resource that is responsible for displaying information, such as the image and text. The detailed

categories are defined in the table 5.2 of the Chapter 5. The real challenge lies in the significant diversity of individuals in the same group, especially for image components, as shown in Figure 3.1(a). That is, the contents of image elements can be literally everything. For example, a selfie can be put on the interface as an image component, while a group picture that contains plenty of people can also be regarded as a single image, as well as a natural scenery photo or even a cartoon illustration can become an image element on the interface. It is also possible that an individual element is a clipping section of an image. In other words, the variation in the same class can be significant, which differs from the conventional object detection tasks that identify more or less similar targets as the same class.

Property 2: The components on a user interface are picked. A variety of components might be allocated in a compact layout, and they would also overlap with or superimpose on others. It is common to put some buttons and text on the background image where the colourful and various contents are displayed. Besides, some independent elements sometimes should be treated as a whole. For instance, a special object named image button is widely used in the mobile application interface, in which a piece of text is placed in the centre of a background image in shape of rectangle or oval. Detection, in this case, requires accurate component segmentation that does not separate the integrated objects while identifies it as a whole. In addition, some mobile applications that work on a small screen are designed in a tight style where the elements are close to each other, which raises the difficulty for precise localization of UI components.

Property 3: The user interface component's shape is arbitrary. This property is especially for the width, height and aspect ratio. Although there are some rules for user interface design in terms of the size and shape, as shown in Figure 5.10, the elements' character in the same class can still vary to a large degree. As the first attribute, this variation poses a great negative influence on the accuracy of localization, because most deep learning based object detection methods achieve localization by bounding box regression [Girshick et al., 2014b; Lee et al., 2019]. Regression essentially is a statistical approach modelling the relationship between a dependent variable and one or more explanatory variables [Freedman, 2012]. For example, in linear regression, the relationships are modelled by the linear predictor functions estimated from the known data. However, if the data is too dispersed, the estimated functions are hard to be accurate and effective [SEAL, 1967]. Thus, localizing methods that rely on the bounding box regression is not robust in human-computer interface components detection.

Property 4: The position and boundary of components in a user interface demand of absolute precision. In web and mobile application development, developers implement the components in the way that sets the accurate size (width, height or aspect ratio) and places them in an exact position (pixel distance from the boundary). This character asks for as accurate the detection of components on the interface as possible. But as mentioned in the third property, the bounding box regression that deep learning based object detection methods use is a statistic estimated function, which is inadequate to predict the one hundred per cent precise result.



(a) Heterogeneous contents of images on web page (b) Desirable detection result

Figure 3.1: A real web interface design. The 3.1(a) represents a common case in human-computer interface design. Various colourful and heterogeneous images are used to display information, in which the contents can be everything and may confuse the neural networks. Figure 3.1(b) is the result of my approach using inventive image processing algorithms.

Therefore, the component detection in this case requires some domain-specific approaches that adapt to the particular properties of the user interface. Since the analysis through the data and the popular deep learning methods draws the conclusion that the end-to-end neural networks do not well fit into this mission, I propose a technique that combines with some conventional computer vision and image processing algorithms to accommodate the specified characters in the artificial interface design. Before introducing my own approach, comprehending the mechanisms of deep learning based object detection techniques are critical to bypass their defects.

3.2 Deep Neural Network's Mechanism

Although the deep learning approaches are dominated in the contemporary computer vision field, some natural deficiencies of those techniques still exist, and the flaws indeed cause some issues. Some of the defects are exposed in the mission of human-computer interface component detection, as stated previously. I dig into several popular object detection methods to try to summarize their mechanisms and attempt to reveal the inadequacies.

Generally, two directions of object detection techniques are mostly studied, the region proposals (RCNN, Fast RCNN, Faster RCNN) and single-shot methods (SSD and YOLO) [Gandhi, 2018]. Those approaches are proven effective in natural scenes, and they are so inspiring that a wide variety of derivative techniques refer to them. This section briefly summarises their principles and analyses the flaws that prevent them from performing well in human-computer interface components detection.

3.2.1 Region-based Methods

The reason why we cannot directly build a classic CNN followed by a fully connected layer to proceed object detection is that the length of the detection's output is not fixed [Gu et al., 2018]. That is, the number of occurrences of targets is variable. To address this issue, an idea is that selects various regions of interest from the input image and classifies them respectively with a CNN to inspect the presence of objects in the regions [Gandhi, 2018]. The region proposal approaches derive from this idea and adopt some strategies to alleviate the computational problem that the huge number of regions in different spatial locations and various aspect ratios.

Region-based Convolutional Neural Network: To solve the problem of selecting a huge number of regions, Ross Girshick et al. proposed a technique that utilizes the selective search [Uijlings et al., 2013] to extract smaller amount (always 2000) of regions from the input image. They are named region proposals. Thus, the number of regions that need to be classified reduce to 2000. Then, those regions are resized and thrown into a convolutional neural network followed by a dense layer of 4096 in size at the end. The network hence outputs a 4096-dimensional feature vector for each region. Those vectors are fed into a pre-trained Support Vector Machine (SVM) to classify the presence of objects. The RCNN uses the bounding box regression [Felzenszwalb et al., 2010] to increase the accuracy of object localization.

Several problems are existing in the RCNN. First, although it reduces the number of regions, it still has to classify 2000 region proposals. Second, as the long-time (47s average) it takes to process all regions, it can hardly be used real-time. Third, it locates objects by using bounding box regression, but as mentioned in the last section, this method cannot predict the actual outline and precise location of objects but just perform statistical regression.

Fast RCNN: Ross Girshick et al. then proposed the Fast RCNN to solve some of the drawbacks of the previous version RCNN. The improvement of this approach is that it builds a CNN to generate a convolutional feature map by feeding the input image, instead of processing the thousands of region proposals every time on the original image. The Fast RCNN uses a region of interest (RoI) pooling layer to recognize the region proposals from the feature map and resizes them into squares to be the input of the final fully connected layer. In the end, those RoIs are turned into feature vectors, which then be fed into a softmax layer to predict the classes of regions, as well as inputted into a bounding box regressor to localize the accurate

positions of objects.

Although the Fast RCNN makes some improvements, especially in terms of the processing time, the essential mechanisms are similar to its predecessor. They all try to propose regions by some means and identify the presences and classes of objects in these regions. Meanwhile, they all utilize a bounding box regressor to predict the precise locations of targets.

Faster RCNN: Both the two previous techniques apply the selective search to pick region proposals. One drawback of the selective search is that it is a time-consuming process, as well as a fixed algorithm which can not learn for itself like the neural network. In order to bypass this shortcoming, the authors of the Faster RCNN proposed a region proposal network (RPN) [Ren et al., 2015] that is capable of learning the region proposals. Similar to the Fast RCNN, this method uses the CNN encoder to generate feature maps from the input image. Then the RPN is applied separately to predict the region proposals, which followed by a non-maximum suppression (NMS) [Canny, 1987] to refine the resulting predictions.

Therefore, we can observe some common grounds from the series of region-based approaches. One salient similarity is that they all propose multiple regions and conduct prediction and regression on those regions independently. On the other hand, another branch of deep learning based object detection approaches try to process the input image as a whole other than focusing on the local section of the image, and they complete the detection in a single convolutional neural network. To some degree, they convert the detection problem into a regression problem, and hence also are called regression-based methods [Zhang et al., 2019].

3.2.2 Single Shot Methods

This section analyses two representatives of the regression-based approaches [Zhang et al., 2019], the You Look Only Once (YOLO) proposed by Redmon et al. and the Single Shot Multibox Detector (SSD). They all perform object detection by using a neural network to an entire image and pose the detection problem as a regression problem [SACHAN, 2017].

YOLO: Essentially, the YOLO achieves detection by learning the class probabilities and the coordinates for each bounding box. This technique splits the input image into S x S grids, in which each grid cell has B bounding boxes. The outputs of this model are class probabilities and position coordinates of the bounding boxes. Then those whose probabilities are above a threshold are chosen and used to localize the objects within them. In the training stage, if the centre of a ground truth object falls into a grid cell, this cell is responsible for detecting it in the way that finds the bounding box with the maximum intersection over union (IOU) over the true object.

SSD: The single shot multibox detector, as its name suggests, is also a single-shot

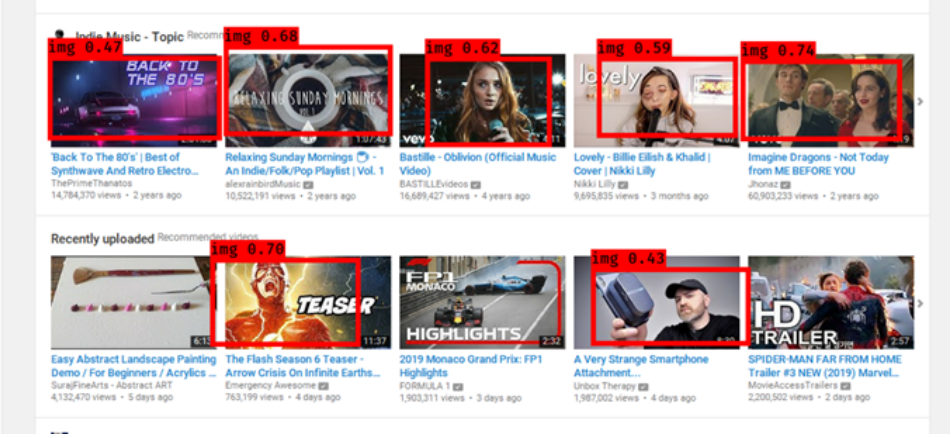


Figure 3.2: The detection result of YOLO. The predicting images are labelled in red bounding boxes, from which we can observe that the accuracy of the localization cannot fit the strict requirement in real User interface design, as stated in *Property 4*. The main reason for the deficiency is that the localization is achieved by the regression of the offset values of bounding boxes.

detector for multiple categories which is faster than its previous state-of-the-art technique. The speed advantage mainly comes from the design that does not resample the bounding box or generate proposals as the RCNN series do. The improvement in accuracy attributes to a combination of predictions of multi-scales feature maps with various resolutions, which handles the problem of variation in object's size. This method still uses a fixed set of default bounding boxes, and it applies small convolutional filters applied to feature maps.

In summary, the object detection task consists of two major parts, the localization of the targets and the classification. As mentioned at the beginning of the section 3.2.1, the key difficulty lies in how to find the objects accurately and efficiently, while the classification is already well handled by CNN. To this end, a variety of ideas have been proposed, and two directions are most widely studied among them, the region-based and the one-shot methods.

The primitive region-based approaches, such as the RCNN and the Fast RCNN all adapt the selective search algorithm to propose potential regions where objects might be. However, some of their drawbacks prevent them from working well in the human-computer interface. First, the selective search depends on the similarity of regions, while the components in an interface can be heterogeneous as the *Property 1* analyzed in section 3.1. Thus the effectiveness of the selective search is undermined. Second, because of the *Property 1* and *Property 2*, it is very easily that the components are oversegmented, especially for the image components that contain colourful and various contents. Oversegmentation is a common issue for region segmentation methods, which means integrated objects are segmented into multiple sub-components by mistake [EGGLESTON, 2015]. Consequently, the single image

component would be split into many sections that will be counted as false positive.

The Faster RCNN avoids using the fixed selective search algorithm and turns to a learnable RPN to propose potential regions. In detail, the anchor boxes are responsible for presenting the regions and fed into regression layer and classification layer. However, as the analysis in the *Property 3* and *Property 4* of human-computer interface, the huge variances of shape in the UI components hamper the effectiveness of logistic regression, while the highly precise localization is required.

Same problems exist in the one-shot methods. Because they are fundamentally regression based approaches, they still suffer the issues brought by the *Property 3* and *Property 4* when applied in the artificial interface. In addition, the oversegmentation is also a challenge for all methods using the default bounding box hypothesis. The heterogeneous contents of the image element always affect the object judgement of the bounding box and produce misrecognition, such as improperly identifying a part of an image as an individual interface component.

3.3 Experiments and Comparison

I conducted an interesting experiment to prove the insufficiency of the object detection approach. YOLOv3 is used to predict the position and bounding box of a pure rectangle without any image contents, while the performance is unsatisfied as expected.

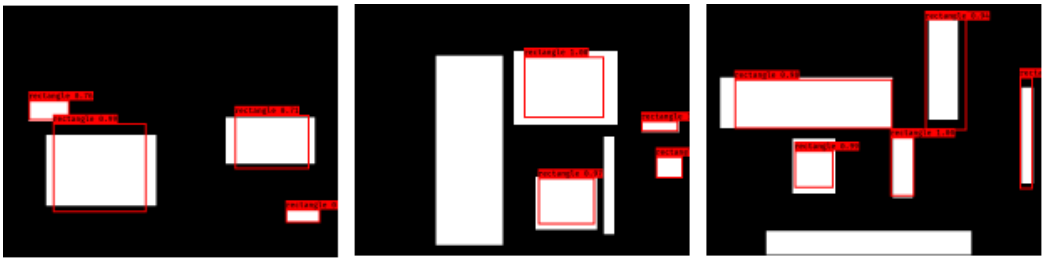
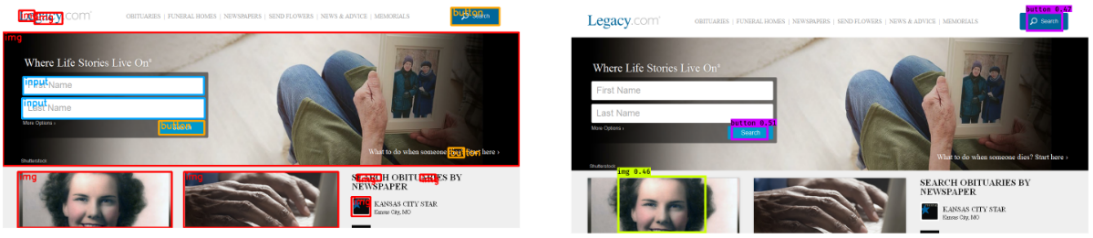


Figure 3.3: YOLO prediction for pure rectangles without contents. The white rectanges are labeled as positive sample (objects), and the red bounding boxes are detection results of YOLO.

Figure 3.3 demonstrates the results of detection for simple shapes. I intentionally removed the content of image and only left the pure white rectangles. However, even for these seemingly clean objects, the object detection approach cannot guarantee accurate prediction.

Due to a lack of time, I did not reproduce all the aforementioned deep learning based approaches but just compared my method with the YOLO to illustrate the result. This YOLO model is on the basis of VGG16, I retrained the last ten layers of this network through totally 13230 images with five different UI component classes refer to table 5.2. Several results are visualized below:

(a)



(b)

(c)

Figure 3.4: Three pairs of comparisons between the detection results of UI2CODE (left hand side) and the YOLOv3 (right hand side)

3.4 Summary

We summarised four particular characters of the UI data we aim to process and the mechanisms of the state-of-the-art deep learning based object detection techniques. This chapter thoroughly explores the common properties of the machine learning methods, in particular, the bounding box regression, and draws the conclusion that the statistic estimation can hardly predict the one-hundred per cent accurate boundary of a component. On the contrary, the seemingly primitive image processing algorithms fit well into this task. Several reasons related to the unique characters of the artificial UI cause this result, including the pixel-level processing and the insensitivity to complex contents. I will elaborate on them in chapter 5 with technical details.

Chapter 4

Data Collection

The initial step of this work is to inspect related data and to build the datasets for further analyses and experiments. Particularly, to meed the specific requirement of this project, the datasets should consist of a variety of images of graphic user interface (GUI) designs, including both website data and mobile application data. Thus, this chapter introduces the methodologies and issues in the data collection process, as well as the utilization of some published datasets.

4.1 Web Dataset

The objective of this project is to detect the components in the human-computer interface or user interface (UI). The general input data should be an interface design image, such as a screenshot of a real web page. Therefore, I collected such screenshots by crawling over a variety of websites to build the web page dataset; the element's annotation is also gathered in this process if possible. In detail, the annotation is directly fetched from the source code of a website, which consists of the tag name (class), size (width and height) and location (coordinates) of the element. However, various issues occurred and made this process far more difficult than it looks at first glance. This section presents those problems and gives solutions to settle or bypass them.

4.1.1 Dataset Construction

I designed a system that automatically mines the screenshots of websites by using a Python-based web application testing framework, Selenium. Generally, the web crawler is chosen to collect data from numerous websites, but the drawback of conventional crawling agent hampered me from doing so.

Web crawler stands for a program script that systematically browses the Internet. It usually takes a list of initial links (URLs) as the so-called *seeds*, [Kobayashi and Takeda, 2000] then visits them one by one and downloads the web page for further analysis. The downloaded web page is also known as the document object model (DOM) with a tree structure [Whitmer, 2009]. All kinds of web component information, including the hierarchy, size and location of the element can be retrieved by parsing the DOM tree.

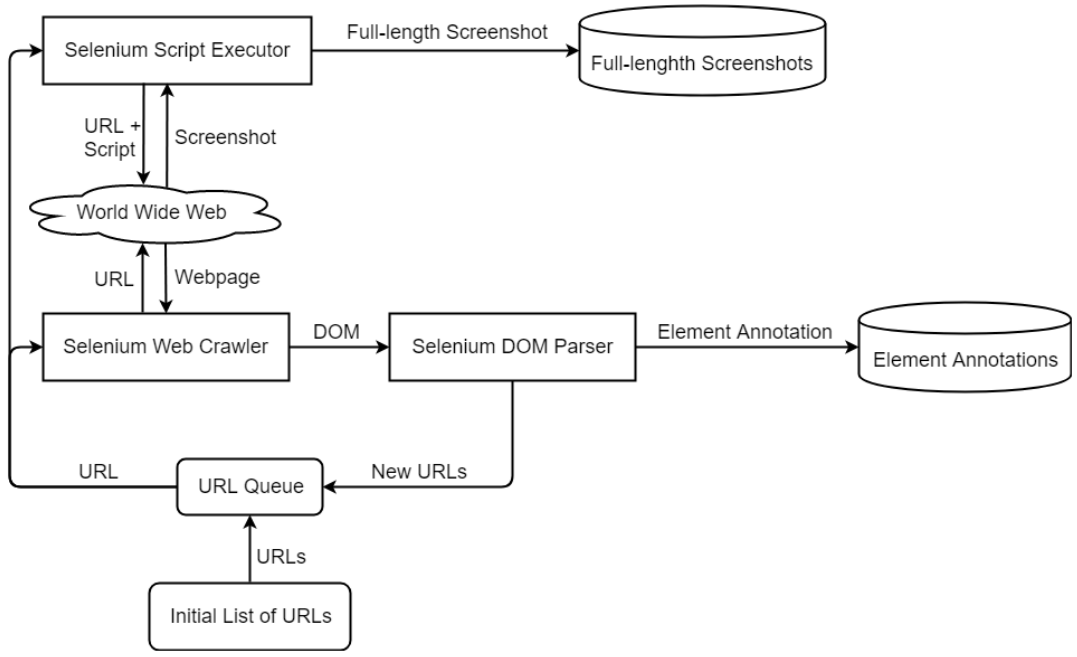


Figure 4.1: The architecture of the data collection system. The initial list of URLs are pushed into the URL queue first, and then the Selenium Web Crawler and the Selenium Script are fed with a new URL pop up from the queue. The Script Executor runs the script to scroll over the web page on the World Wide Web and generates a full-length image. Meanwhile, the Crawler downloads the DOM file of the web page and conveys it to the Parser to retrieve the elements' annotations. In the end, the dataset consisting of screenshots and corresponding annotations is generated.

A common limitation for web crawlers, such as the *BeautifulSoup* [Richardson, 2015], is that they can only conduct parsing on the basis of the static documentation, while they do not support interaction with the website. In other words, they work by means of downloading the DOM and then analyzing the DOM file without any communication with the web. But the data we desire is the full-length screenshot of pages, which cannot be directly obtained by normal crawling agents. Therefore, I turned to the test automation [Moizuddin, 2019].

4.1.1.1 Selenium

Originally, test automation in software testing is defined as the using of software separate from the tested one to execute a series of commands and compare the actual outcome and the predicted outcome [Huizinga and Kolawa, 2007]. In our case, I leverage this feature to execute the script that scrolls down the page to the bottom to obtain the full-length snapshot.

Selenium is a software developed for web test automation. It is composed of several components for aiding the development of web application test automation.

Its own independent integrated development environment (IDE) is implemented as a Chrome Extension, which supports recording, editing and debugging of web application tests. Selenium also has some APIs for various programming languages such as Python and Java. In order to convey messages between the browser and the client, Selenium developed the *WebDriver*, a module accepting the commands via the API and passing them to the browser. The WebDriver is specific to browsers in the way that it sends specified instructions to them to run and retrieves the results. [Moizuddin, 2019].

Selenium can well perform the full page screenshot capture by executing the scripts, as well as the DOM retrieval by web crawling. Besides, it has been implemented as a package in Python, which is convenient to be incorporated into the whole UI2CODE system. Hence I built the data collection module by it.

4.1.1.2 Breadth-first Search

Generally, there are two strategies for web crawling, the breadth-first search and the depth-first search [Kobayashi and Takeda, 2000]. The depth-first search in this context is the common practice for crawlers. In this way, the crawling agent visits the first initial URL and retrieves all the new links on this page. The new links are pushed into the top of a stack. After completing the process of the current page, the next URL is pop out and parsed, and the new links on that site are pushed into the stack again. This process repeats until the stack goes empty or reaching an end condition. On the other hand, the breadth-first means visiting through all the initial list of URLs first and adding the new links fetched from the crawled websites to the end of a queue. The new links will not be accessed until the given ones are all visited.

The breadth-first search strategy is adopted. We want to collect interface designs as varied as possible so that the components detection techniques can be thoroughly tested and developed. Thus, the crawler does not go deep of a single website but visits various websites as many as possible.

In summary, the data collection system is composed of three modules: the web crawler responsible for DOM file downloading from the given link; the parser takes the DOM file as input and parse it to retrieve element annotations, such as tag, size and position; the script executor run a piece of JavaScript code to scroll through the web page to take the full-length screenshot. The entire system is demonstrated in Figure 4.1.

4.1.2 Problems with Web Crawling

Several issues exist in this process caused by the malfunction of Selenium and the intrinsic ambiguity of the front-end language [Basten, 2011; Riva, 2019]. This section presents two kinds of errors of collected data, these faults, unfortunately, can only be bypassed but cannot be resolved perfectly yet.

4.1.2.1 Ambiguity of Web Components

One inevitable problem is that there can be various ways to present or implement the same component in web user interface [Riva, 2019]. This causes the component's ambiguity that the class of a UI component can be uncertain. For example, a button on the web page can be implemented by the HTML tag <*button*>, while a variety of other HTML tags, such as <*div*> and <*a*> with particular style setting or attribute, can perform the identical effect.

Therefore, it is inadequate to label the components according to their HTML tags simply, instead, the style setting (CSS) and functionalities (JavaScript) should also be taken into account.

One rule we follow in the data collection process is that we label the components with the same visual effect as the same class to avoid confusing the classifier. Thus, in accordance with the defined UI component category in table 5.2, we group some HTML tags that provide a similar function together and make a table as below:

UI Component	HTML-Tag	CSS-Style	JS-Functionality
Button	$\langle button \rangle$	-	-
	$\langle a \rangle$	$display : block$	-
	$\langle div \rangle$	$display : block$	$onClick = function()$
	$\langle input\ type = "button" \rangle$	-	-
Input Box	$\langle input \rangle$	-	-
Image	$\langle img \rangle$	-	-
Icon	$\langle i \rangle$	-	-
Text	$\langle p \rangle$	-	-
	$\langle span \rangle$	-	-
	$\langle h* \rangle$	-	-
	$\langle a \rangle$	$text - decoration : none$	-
Block	$\langle div \rangle$	$border : solid$	-

Table 4.1: Some identical UI components can be implemented by different HTML tags with specific CSS style and JS functionality. A components will be labeled with that class name as long as they meet all the three requirements at a row. The $\langle h* \rangle$ means the tag can be $\langle h1 \rangle$ or $\langle h2 \rangle$ or $\langle h3 \rangle$. The "-" signifies that no particular requirement for that factor.

However, some developer may not explicitly write those features in HTML or CSS. They would set layout attributes in the JS function that dynamically adjusts the parameters. In this case, the Selenium cannot directly parse the DOM file to retrieve the correct properties of elements. Unfortunately, this issue can hardly be addressed by algorithm because there is no universal standard to follow for web development. Therefore, we have to manually check the correctness of the collected data, which is an obvious limitation for this system now.

4.1.2.2 Malposition of Annotation

While the Selenium is undoubtedly a powerful and convenient test automation tool, it has some errors exposed when retrieving the annotation from the web page. In detail, the positions of elements are not accurate sometimes, and they always deviate a bit from the real position.

A difficulty to settle down this problem is that this phenomenon is non-deterministic. In other words, the system works well for some websites while dysfunction in others. For example, it may retrieve the correct annotations of the corresponding components for half URLs, but it also produces offset for some links, as shown in Figure 4.2.

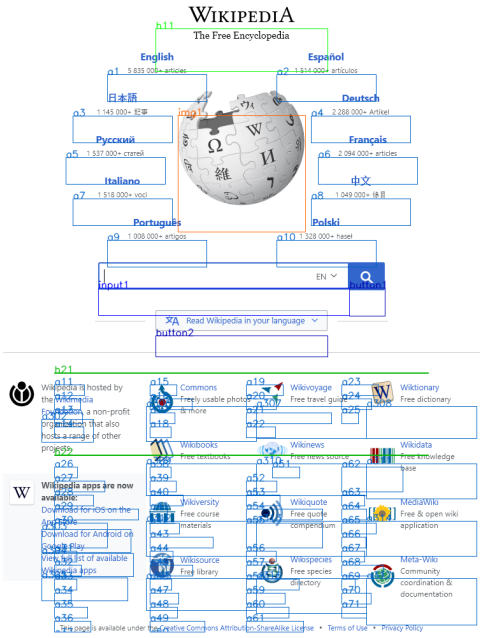


Figure 4.2: The Selenium sometimes produces wrong annotations mismatching with the corresponding elements. This problem is intermittent and unpredictable, which causes it difficult to be fixed by certain method.

The further analyses and experiments suggest that this issue roots in the defect of Selenium Webdriver. In addition, because of the unpredictability of the offset, it is also impractical to set an adjusted value for all retrieving annotations. Therefore, it is unreliable to count on the automatic system totally, and the handpicking should be involved.

4.2 Mobile Application Dataset: Rico

Other than webpages, another major branch of the graphical user interface design is the mobile application interface. There are some differences between these two cases, because of the diverse design requirements and working environment of the applications. Therefore, we leveraged a mobile UI dataset to fill in the gap, an existing

dataset of real working apps named Rico [Deka et al., 2017].

The means of collecting screenshots of working mobile apps is particular compared with the previous work of web data. Understandably, mining mobile applications differs from the practice of crawling websites because of the disparate underlying implementations, and we can hardly reuse the scrawling tool created for web in mobile apps. Fortunately, researchers have already done various works and published several high-quality datasets [Deka et al., 2016; Sahami Shirazi et al., 2013; Alharbi and Yeh, 2015]. This work mainly uses a large repository of mobile app designs name Rico [Deka et al., 2017] as the app dataset to conduct experiments and test the UI2CODE.

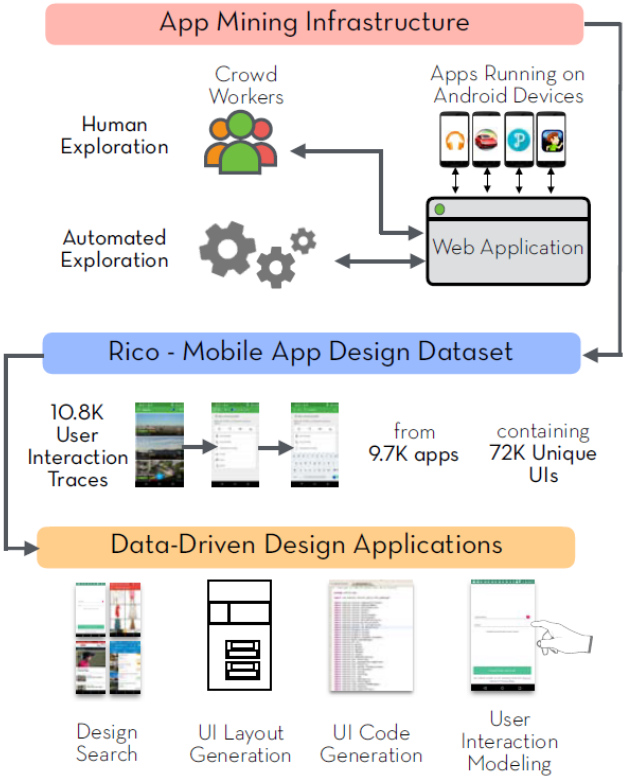


Figure 4.3: Rico combines the human-powered and programmatic exploration to conduct data mining in mobile apps, building a large dataset comprising 72k unique UI designs.

Rico comprises diverse screenshots of real mobile apps, and it includes the visual, textual, structural and interactive properties of UIs. It was designed to support the data-driven models that help designers catch best practices and trends, which can also well fit into our task. The Rico dataset contains design data from more than 9.7k Android apps spanning 27 categories, and totally 72k unique UI screens were recorded. The diversity and quantity of data in Rico gave us a good chance to investigate the design characters of mobile apps to inspire our own system.

Rico was built by mining Android app at runtime in the way that combines hu-

man intervention and automation. One problem in mining app data is that the automated crawler is often stymied by complex interaction sequence required by applications, such as logging in and filling in a form. Therefore, Rico combines the strengths of human and programmatic crawler: leveraging human power to unlock the hindering and applying automated agents to activate the interactive elements to discover more states exhaustively. Besides, the crawlers adopt a novel *content-agnostic similarity heuristic* to efficiently explore the UI state space. Figure 4.3 shows the workflow of Rico.

4.3 Artistic Design Drawing

Sometimes, design drawing can be artistic rather than precisely performing the visual effect of working product. Those images aim to provide the development team with inspiration, but great gap exists between conceptual drawings of graphic artist and the real user interface. Usually, those fancy images either cannot be identically implemented into UI by or takes a long time to do so, because manually creating UI components and adjusting their sizes and spacial positions with programming language are not as directly as drawing. Therefore, we hope to find a way to ease the process of converting an artistic drawing to UI code by using an automated tool.

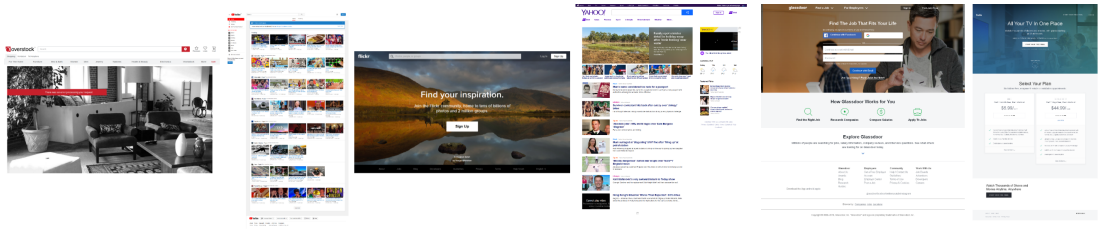
Prior to developing any specific technique to fit this task, we first investigated the target data. To this end, we collected various design drawings from Dribbble, a popular website where designers post their creative works as images or videos. Then the dataset of artistic UI design was used to develop methods and evaluate their performance.

Several issues occurred in this process. First, designers uploaded plenty of works were in video format, which is demonstrative to present their creation, but they are not suitable for our use case. Second, some drawings only focus on fancy illustrations instead of the layout of UI. Third, some images contain a sequence of UIs to show the workflow. To keep efficiency, we simply ignore videos while collecting data from dribbble.com, and we manually filtered out images that are overly fancy and have few information of UI layout.

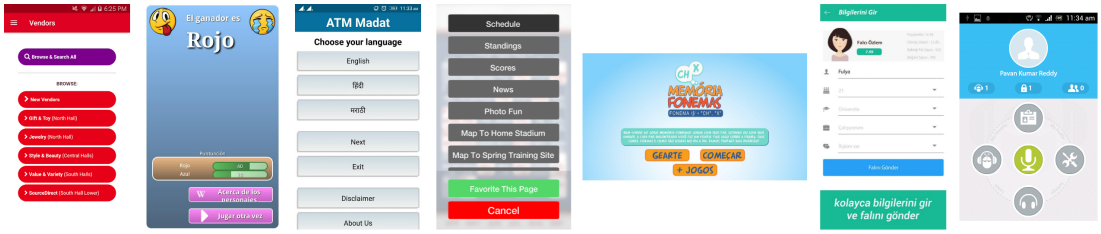
One drawback of this dataset is that there is no corresponding annotation for the images as that for crawled web data because they are pure design drawings that are expected to present aesthetics. Thus they cannot be used to evaluate the effectiveness of detection directly.

4.4 Glimpse of Data

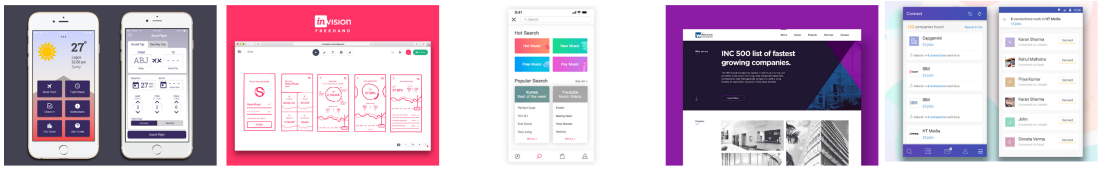
This section presents several examples of each dataset. This thesis only demonstrates a few of typical instances of those datasets, but diversity inner each repository is vast, and we can hardly find a common pattern of the UI design of web or mobile application, which is also why the human-computer interface component detection is worth deeper thinking.



(a) Web Snapshot



(b) Rico



(c) Dribbble

Figure 4.4: Several examples in three datasets.

4.5 Summary

To investigate the human-computer interface, this work built a web UI dataset and an artistic drawing dataset, and it leveraged an existing sophisticated UI repository of working mobile apps. The problems of intrinsic ambiguity of front-end programming languages and malposition caused by Selenium's defect were exposed and overcome in the process. In the end, three datasets containing UI images of web and mobile applications were ready for development and experiment of the UI2CODE.

Chapter 5

User Interface Components Detection

UI2CODE is a system automatically recognizing the semantic contents of the input image of human-computer interface and generate the front-end code that can implement the same visual effect and expected functionalities of the given image design. To this end, I propose a precise and purpose-built user interface components detection pipeline that works particularly better than popular objection detection methods [Ren et al., 2015; Redmon et al., 2016; Redmon and Farhadi, 2018] in the graphical human-computer interface (GUI).

This technique utilizes computer vision and image processing algorithms to detect and locate objects. After selecting the candidate regions where are likely to be UI components, a classifier is built to recognize those areas and categorize them into different classes, such as *button, input bar, images and text*. Meanwhile, the text recognition is achieved by an effective Optical Character Recognition (OCR) technique, the Connectionist Text Proposal Network (CTPN) [Tian et al., 2016]. In the end, the results from those modules are merged and refined to produce the final result.

Performance of the image processing technique compared with machine learning methods, especially in our task, is significantly better in terms of accuracy of components position detection and recall. Detailed reasons for those strengths are stated below, one of the most interesting interpretations is that the deep neural network observes the texture rather than the shapes of objects [Geirhos et al., 2019; Cepelewicz, 2019; Geirhos, 2018], which arouses some deep-going thoughts about the nature of deep learning methods, and I elaborate those thinkings in Chapter 3.

5.1 Architecture

I assume the input to this tool to be an image of an interface, which can be either a screenshot of a real application (web or mobile) or a conceptual design drawing. We focus on segmenting and classifying the possible human-computer interface components on the image.

Based on the common practice of developer and the properties of the front-end programming languages, three categories including totally five types of graphic user

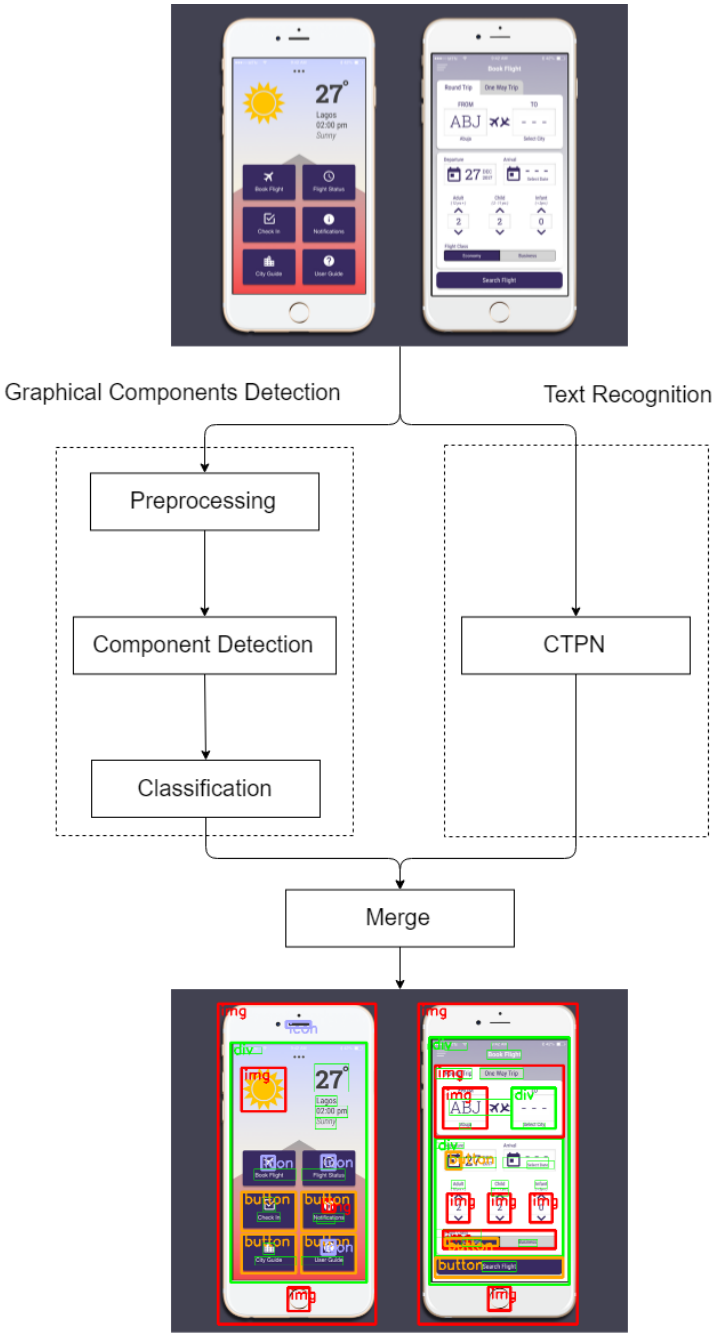


Figure 5.1: The architecture of the UI components detection pipeline. The input image here is a conceptual design drawing of a mobile application, and it is processed by two independent branches to segment the UI components and detect the text regions respectively. Then the results are merged and refined to get the final result.

interface components are defined in this process: interactive elements *(button, input box)*, static resource *(image, icon)* and layout structure *(block)*, see Table 5.2.

To perform precise segmentation, I implement this tool as a three-phase pipeline consisting of pre-processing, components detection and classification, combined with text detection achieved by CTPN. The results from both branches are then integrated at the end to produce the final prediction. The illustration of the overall architecture is shown in Figure 5.1.

5.2 Pre-processing

The first step for the UI components detection pipeline is pre-processing. It transforms the input image into a specific form that is convenient for further processing. Three substeps are conducted here: grey-scale image conversion, gradient calculation and binary image conversion.

In our task, we treat all contents on the image as parts of individual components without consideration of their own detailed information. For example, an image on an interface design should be regarded as a single element instead of a combination of the real contents in it, as shown in Figure 5.2.

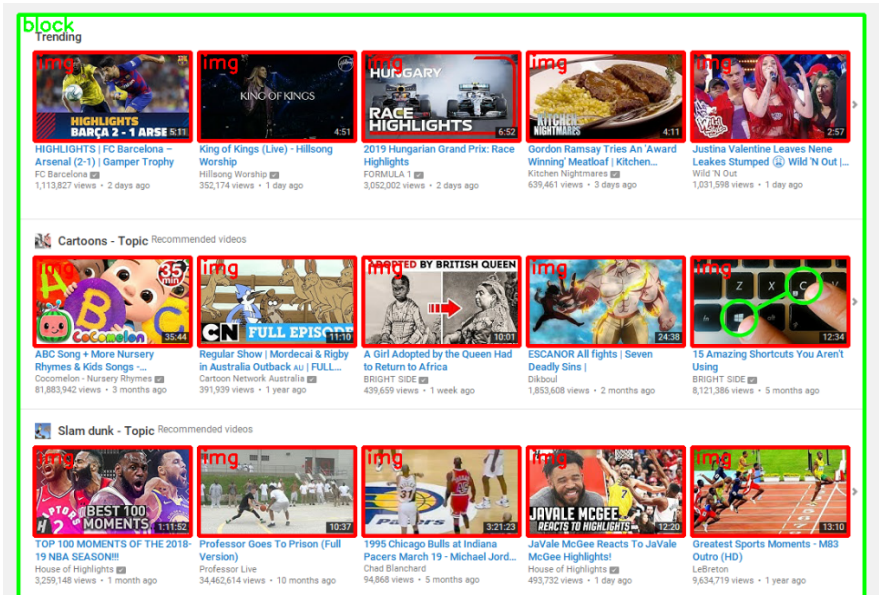


Figure 5.2: A section of screenshot of YouTube website. Plenty of image components (labeled in *img* with red bounding boxes) with colourful contents appear on this user interface, but we want to leave out the information of the real contents and treat them as parts of individual UI components.

To this end, I try to find a means to convert the colorful and complicated image into a simple form that does not contain redundant information this task does not need and is convenient to segment components. The popular related algorithms, such as Canny edge detection [Canny, 1986] and findContour method in OpenCV

based on techniques proposed by Suzuki et al. [Suzuki and be, 1985; Team, 2012], do not work well in this case, because those processing always leave the texture details and disconnect the contents in an image, as shown in Figure 5.3. So, I propose a new method to satisfy this purpose.

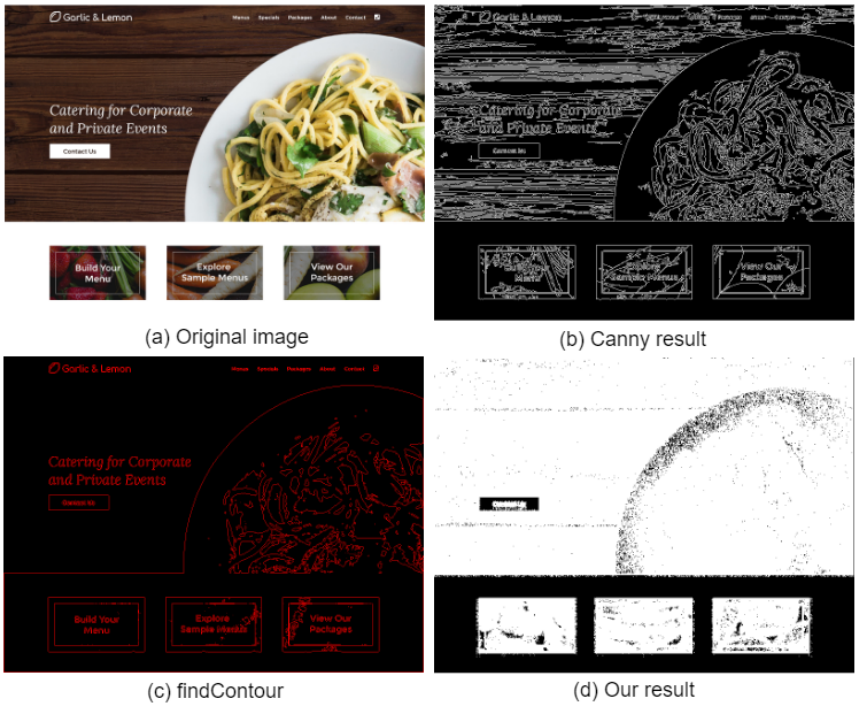


Figure 5.3: The picture (a) is the original image; the image (b) is the result of Canny algorithm, which extracts too many details of texture; picture (c) is the result of findContour function in OpenCV library, and it focuses on calculation of the boundary of objects; (d) is the binary image processed by our method, which converts the components to a simple binary image consisting of few integrated objects without too many redundant texture information.

5.2.1 Gradient Calculation

The gradients of a digital image measure how each pixel changes in terms of significance and direction [Jacobs, 2005]. Popular techniques of image gradient calculation are Roberts cross operator, Prewitt operator, and Sobel operator [Roberts, 1963; Prewit, 1970; Sobel, 1968]. We can acquire two pieces of information from the gradient of each pixel, the direction of the change and the magnitude of this change.

However, unlike other common computer vision tasks that deal with the natural scene, we do not care for the changing direction as much as about the magnitude, because we focus on determining whether a pixel is a part of the potential components rather than the detailed information of how it changes. Therefore, we calculate

the magnitude of the gradient by formulae below:

$$gx = \frac{\partial f(x,y)}{\partial x} = f(x+1,y) - f(x,y) \tag{5.1}$$

$$gy = \frac{\partial f(x,y)}{\partial y} = f(x,y+1) - f(x,y) \tag{5.2}$$

$$G(x,y) = |gx| + |gy| \tag{5.3}$$

where: f(x,y) is the pixel value for point (x,y) in the image; gx and gy are the gradients in the x direction and y direction respectively; G(x,y) is the magnitude of gradient value at pixel point (x,y).

The result of this step is a grey-scale map (a two-dimensional matrix in which the value of each pixel is on the scale of 0-255) reflecting the significance of gradient of the original image, as shown in Figure 5.4(b).

5.2.2 Binarization

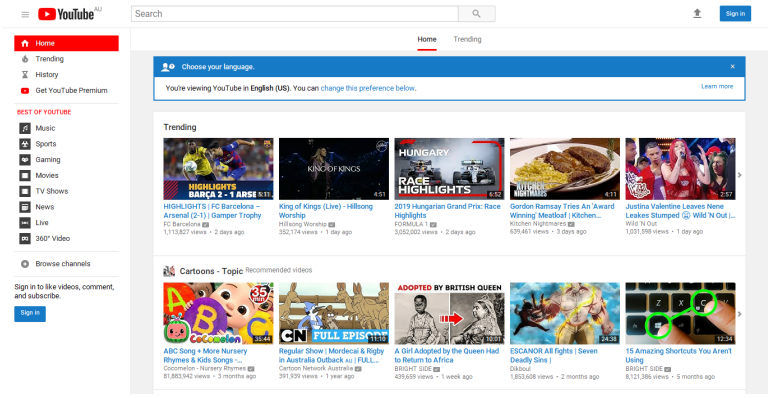
The second substep of pre-processing is to binarize the grey-scale map; the purpose of this process is to intensify the component regions from the background.

One particular observation on GUI that differs from the natural scene is that the regions where there is little or no gradient change are more likely to be the background. On the other hand, pixels with large gradient should be parts of the foreground objects (interface components). With this regard, I set a small gradient threshold to label each pixel as either foreground or background.

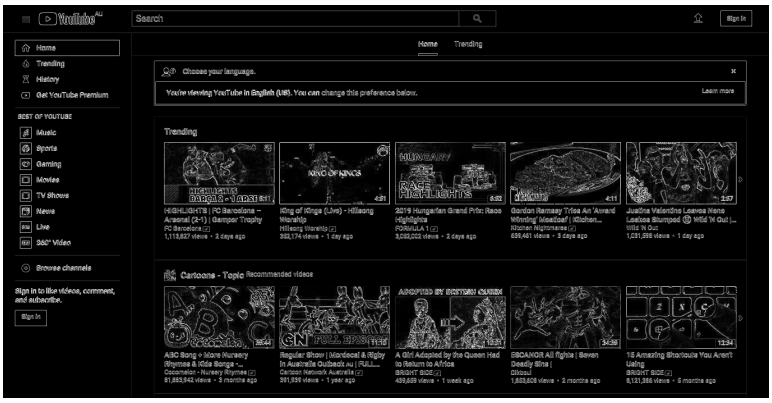
The goal of binarization is to assign a binary value to every pixel in an image [Shapiro, 2002]. T.Sezgin and Sankur summarized the thresholding methods into six categories: the histogram shape-based, clustering-based, entropy-based, object attribute-based, spatial methods, local methods [Mehmet Sezgin, 2004]. My technique is similar to the Entropy-based methods, which use the entropy of the foreground and background regions. Specifically, as mentioned above, the gradients of the pixels in the foreground are usually larger than zero, while the background of GUI is always unicolour and the gradients are zero. In our case, this step assigns either 255 (white) or 0 (black) to each pixel; points whose value is 255 means could be parts of an interface component; value 0, on the contrary, means this point is part of the background, as demonstrated in Figure 5.4(c).

But for different datasets, the gradient property would be slightly different, which requires adjustment of the threshold. For instance, the images in Rico dataset are more compact than the screenshots of real web pages, so the threshold should be slightly higher to better segment regions; and the images of Google Play Poster and Dribbble datasets can be lower resolution than are web pages, so the background is more blurred and its gradient can be higher than zero.

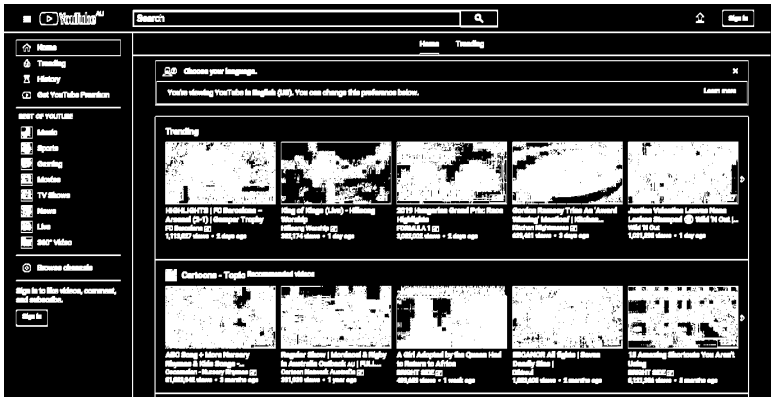
In the end, a binary map that contains clear foreground objects is produced by the pre-processing for further operations.



(a) Original input image



(b) Gradient map



(c) Binary map

Figure 5.4: The visualized demonstration of the pre-processing. The original image Figure 5.4(a) is given as the input; and the process calculates the magnitude of the gradient for each pixel to produce a gray-scale map Figure 5.4(b); then according to the observation of foreground and background in the human-computer interface, a binary map Figure 5.4(c) is generated

5.3 Component Detection

Based on the acquired binary map, this stage tries to extract component candidates and heuristically categorize the connected regions that could be potential user interface elements. To this end, this process contains several substeps: Connected Components Labelling, Component Boundary Detection, Rectangle Recognition, Block Recognition, Irregular Components Selection and Nested Components Detection.

Three sets of objects yield from this process, *Block, Image and Interface Components*. Detailed categories and classes of UI components are defined in section 5.4, at this stage, the connected components are only grouped to three aforementioned general classes according to some heuristic rules based on the size and aspect ratio in order to save processing time in the next step.

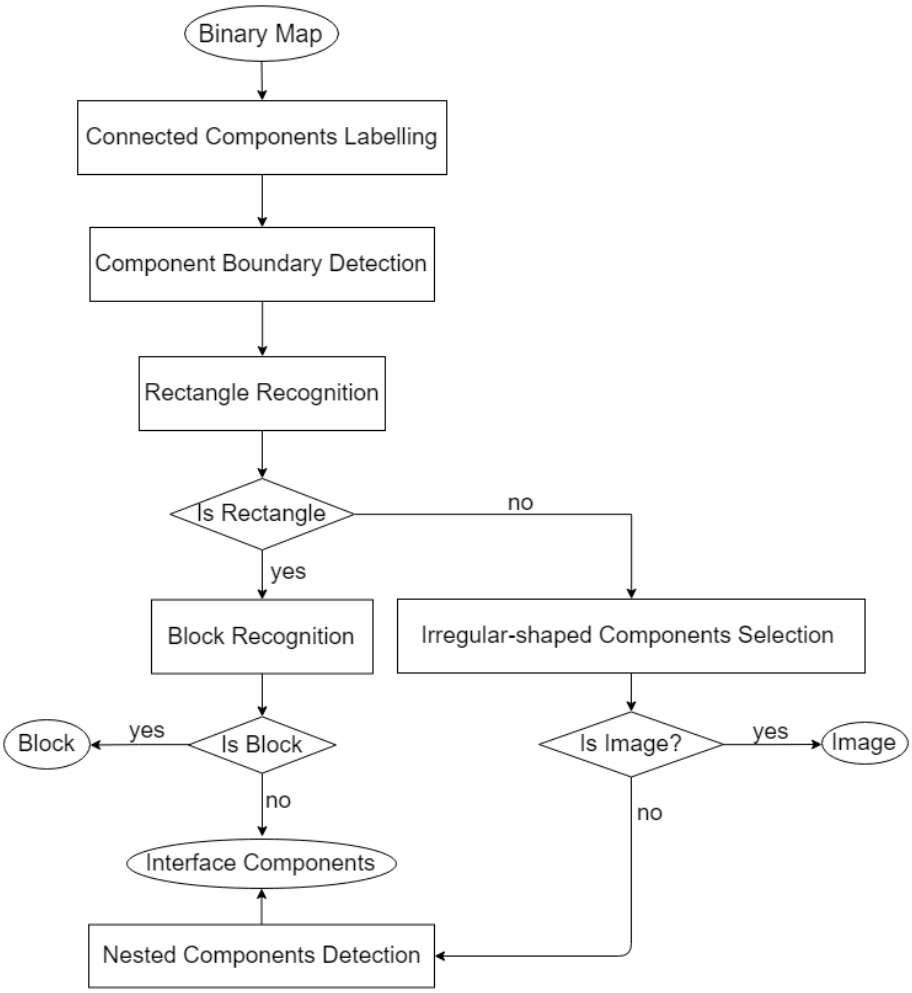


Figure 5.5: Flow chart of the Component Detection pipeline. This pipeline takes binary map as input, and the result of this step consists of three categories of UI elements: *Block, Image and Interface Components*

5.3.1 Connected Components Labelling

This process refers to the connected-component labelling (CCL) algorithm, which is demonstrated in Figure 5.6, the purpose is to assign each pixel a label identifying the connected component to which this pixel belongs [Samet and Tamminen, 1988; Dillencourt et al., 1992]. In other words, this substep aims to segment connected components from the binary image.

I refer to the Seed Filling algorithm in computer graphic [Vincent and Soille, 1991] to implement my own method. The pseudocode of this technique is shown in Algorithm 5.3.1.

Algorithm 1 Connected-component labeling

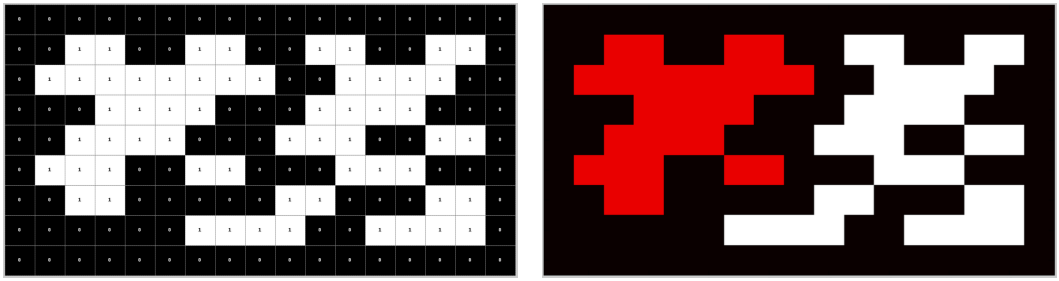
Input: Binary map
Output: An array of components, each component contains a group of points that constitute it

```
1: Components ← [ ]
2: MarkingMap ← Zeros(BinaryMap.shape)
3: for Point in BinaryMap do
4:     if Point is 255 and MarkingMap[Point] is 0 then
5:         MarkingMap[Point] ← 1
6:         Neighbors ← new Queue.push(Point)
7:         Component ← new Stack.push(Point)
8:         while Neighbors.Length > 0 do
9:             NextPoint ← Neighbors.pop()
10:            for Neighbor in NeighborsofPoint(NextPoint) do
11:                if Neighbor is 255 and MarkingMap[Neighbor]is 0 then
12:                    MarkingMap[Neighbor] ← 1
13:                    Neighbors.push(Neighbor)
14:                    Component.push(Neighbor)
15:                end if
16:            end for
17:        end while
18:        Components.push(Component)
19:    end if
20: end for
21: return Components
```

Alternatively, this algorithm can be written as:

(1) Start from the top-left point, initialize a MarkMap with the same shape as the input image and fill it with zeros to indicates if points are already labelled, go to (2).

(2) If this pixel is foreground (its value is 255), and it hasn't been labelled (MarkMap is zero at this position), then add it to a store queue and count it as a point of a new component, and go to (3); otherwise repeat (2) for the next pixel in the binary map.



(a) Binary map (b) Connected-component labelling result

Figure 5.6: Connected-component labelling demonstration. The 5.6(a) shows foreground (white points) and the background (black points); the 5.6(b) is the CCL result, where two connected components are labeled in red and white colour.

(3) Pop out an element from the store queue, inspect all of its neighbours. If the neighbour is foreground and hasn't been labelled, then add this point into the store queue and count it as a point of the current component. Repeat (3) until the store queue is empty, then go to (4).

(4) Scan the next pixel in the binary map and go back to (2) until all pixels in the input image are inspected.

As described in the above algorithm, the output of this function is an array of components. The way I store a component is presenting every connected component as a list of points (coordinates), where the points are all connected foreground in each list.

5.3.2 Component Boundary Detection

For each connected component, I calculate its outer boundary by a four-border method. The resulting boundary consists of four borders: border-top, border-bottom, border-left and border-right.

Because the sole information of a component acquired from the last step is the points that constitute this component, so for the sake of efficiency, I implement this method by means of storing the borders in form of $\{position : boundaryvalue\}$ through dictionary data structure, *position* is key of the dictionary and *boundaryvalue* is value. For instance, the format of points in *BorderTop* and *BorderBottom* is $\{column : maxrow\}$ and $\{column : minrow\}$, which indicates the vertical boundary values of this column in this component. In the same way, points in $BorderLeft$ and *BorderRight* are stored as $\{row : mincolumn\}$ and $\{row : maxcolumn\}$ to show the horizontal boundary value of each row.

In Algorithm 5.3.2, the Borders are initialized as four empty dictionaries that will eventually be filled with points in the aforementioned form. $BorderTop[Column]$ means retrieving the vertical boundary value (minimum row index) at this column of the component, and if this value is larger than the row index of the current point

Algorithm 2 Four-border Boundary Detection

Input: Component consisting of points that belong to it
Output: Boundary containing four borders: top, bottom, left, right; Each border is a list of points

$BorderTop, BorderBottm, BorderLeft, BorderRight \leftarrow \{\ \}, \{\ \}, \{\ \}, \{\ \}$

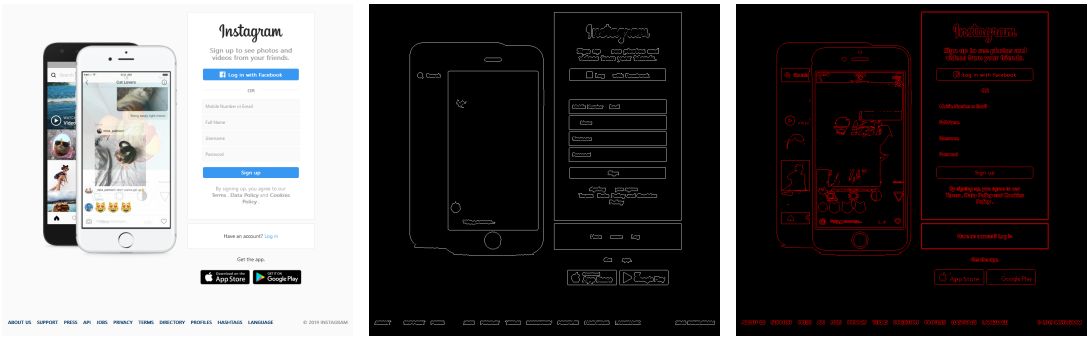
for $Point$ **in** $Component$ **do**
  $Row, Column \leftarrow Point[0], Point[1]$
  if $Column$ **not in** $BorderTop$ **or** $BorderTop[Column] > Row$ **then**
    $BorderTop[Column] \leftarrow Row$ ▷ Choose the smaller row as top
  end if
  if $Column$ **not in** $BorderBottom$ **or** $BorderBottom[Column] < Row$ **then**
    $BorderBottom[Column] \leftarrow Row$ ▷ Choose the larger row as bottom
  end if
  if Row **not in** $BorderLeft$ **or** $BorderLeft[Row] > Column$ **then**
    $BorderLeft[Row] \leftarrow Column$ ▷ Choose the smaller column as left
  end if
  if Row **not in** $BorderRight$ **or** $BorderRight[Row] < Column$ **then**
    $BorderRight[Row] \leftarrow Column$ ▷ Choose the larger column as right
  end if
end for
$Boundary \leftarrow [list(BorderTop), list(BorderBottm), list(BorderLeft), list(BorderRight)]$
return $Boundary$

in the loop, then this point should be considered as the new top value at this column in the component, done by $BorderTop[Column] \leftarrow Row$.

For future convenience, the borders are transformed from dictionaries to list in the way that converts point information stored in the dictionary $\{position : boundaryvalue\}$ to a list $(position, boundaryvalue)$, so that all the points in the borders become the format $(position, boundaryvalue)$, which are better to retrieve and change in further process. This conversion is done by $list(BorderTop)$. Finally, this function returns a list of the four borders in order of top, bottom, left and right.

Unlike other popular contour detection algorithms, especially the findContour function implemented in OpenCV [Team, 2012], this task does not care much about the precise outer borders and hole borders for each object, because the purpose at this stage is to select the potential graphic interface components and filter out those unlikely to be interface elements, instead of acquiring the detailed texture information of each object. Besides, the findContour function is highly sensitive to parameters, which means this method is not robust enough and requires adjustment of parameters for various input images.

On the contrary, the four-border boundary detection algorithm is sufficient and more efficient in this case because it only calculates the outer boundary, and is not overly subject to parameters.



(a) Original image (b) Result of Four-border (c) Result of findCountour

Figure 5.7: Demonstration of the Four-border boundary detection. The 5.7(a) is the original input image from a web interface design; the result of this algorithm is shown in figure 5.7(b), which does not contain the fine grained details of the components but only the outer boundaries; the 5.7(c) shows the result of findContour function in OpenCV library, is detects more precise border of objects but the performance is unstable and sensitive of the parameters.

5.3.3 Rectangle Recognition

Another observation on the human-computer interface is that most of the elements have regular shapes. For example, pictures on a website are always be displayed in rectangle regions, and buttons and input boxes are usually round or rectangular. Thus, a rectangle detection algorithm is introduced as a heuristic process for interface components detection.

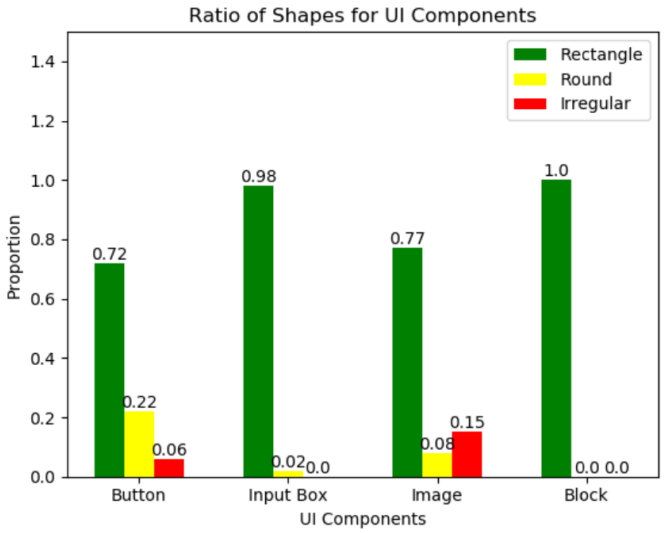


Figure 5.8: The proportion of UI components with different shapes.

The statistics are collected from three different datasets, in which the numbers of *Buttons*, *InputBoxes*, *Images*, *Blocks* are 10566, 3460, 39998, 1568 respectively. The

statistics are consistent with the observation mentioned before; a large proportion of human-computer interface components are in a rectangular shape, which proves the necessity of the rectangle detection process.

Existing techniques, such as approxPolyDP in OpenCV [Dev, 2014] library and Hough transform [Duda and Hart, 1972], are too complicated and rather unnecessary in our task. We only estimate whether the component is a rectangle or not, but the approxPolyDP method based on Douglas-Peucker algorithm [Douglas and Peucker, 1973] involves too much computation to calculate the precise polygonal curves. Hough transform is also too computationally expensive because it examines four parameters to detect a rectangle, which projects the information into a four-dimension computation space.

Therefore, I propose a simple and efficient method to detect rectangle by calculating the smoothness of the boundary. In addition, this method measures the dentation to filter out concave objects. The pseudocode is shown in Algorithm 5.3.3.

Algorithm 3 Rectangle Detection

Input: A component's boundary consisting of four borders: top, bottom, left, right
Output: Boolean value to indicate if this component is rectangular shape

```
1: smoothBorderNo ← 0
2: smootness ← 0
3:
4: for Border in Boundary do
5:     for i in range(Border.length) do
6:         difference ← Border[i] − Border[i + 1]
7:         if different = 0 then
8:             smootness ← smootness + 1
9:         end if
10:    end for
11:    if smoothness/Border.length > 0.8 then
12:        smoothBorderNo ← smoothBorderNo + 1
13:    end if
14: end for
15: if smoothBorderNo = 4 then
16:    return True
17: else
18:    return False
19: end if
```

In this implementation, *Boundary* is an array of size four, which contains four borders for top, bottom, left and right directions. For each border, $Border[i]$ means the ith element in it, and $Border[i+1] - Border[i]$ calculates the variance or gradients between two adjacent points in the same border, which indicates the smoothness of this border.

5.3.4 Block Recognition

We define a bordered region enclosing various multiple elements as a block, Figure 5.9 presents two examples. As explained in section 5.4, the block is a layout structure, which could be regarded as a frame or a box. Block is usually rectangular and hollow, but it is hard to be recognized by the machine learning methods directly because it is often too variant and is easy to be misidentified as an image element, especially when the block containing images as shown in Figure 5.9(c).

However, the simple and general shape attributes (rectangular and hollow) can be well captured by the image processing methods. Therefore, I proposed a block recognition algorithm based on pure image processing technique to identify the block region.

One detailed observation of block is that it can be likened to a wireframe, as demonstrated in Figure 5.9(b) and Figure 5.9(d). In other words, there should be a gap between the borderlines and the contents in it. Therefore, the algorithm identifying block is designed by detecting the gaps, and it only checks the rectangular objects because of the shape attribute. The Python style pseudocode is shown in Algorithm 5.3.4. It just illustrates the basic idea of this algorithm, and the real implementation would be more sophisticated because of the complicity of input images.

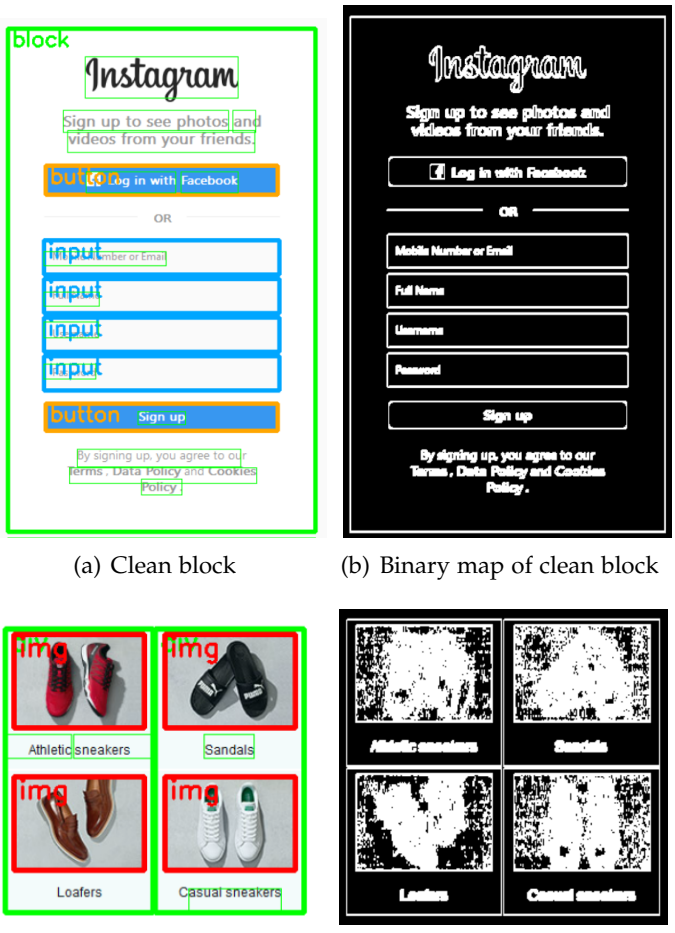
Algorithm 4 Block Recognition

Input: Boundaries of rectangular components; Binary map; Border thickness
Output: Boolean value to indicate if this component is block

$GapTop, GapBottom, GapLeft, GapRight \leftarrow True, True, True, True$
for i **in** $range(1..MaxBorderThickness)$ **do**
    if $\sum BinaryMap[BorderTop + i, BorderLeft + i : BorderRight - i] = 0$ **then**
        $GapTop \leftarrow False$
    end if
    if $\sum BinaryMap[BorderBottom - i, BorderLeft + i : BorderRight - i] = 0$ **then**
        $GapBottom \leftarrow False$
    end if
    if $\sum BinaryMap[BorderTop + i : BorderBottom - i, BorderLeft + i] = 0$ **then**
        $GapLeft \leftarrow False$
    end if
    if $\sum BinaryMap[BorderTop + i : BorderBottom - i, BorderRight - i] = 0$ **then**
        $GapRight \leftarrow False$
    end if
end for
if $GapTop = True$ **and** $GapBottom = True$ **and** $GapLeft = True$ **and** $GapRight = True$ **then**
    return $True$
else
    return $False$
end if

Where the $BorderTop$, $BorderBottom$, $BorderLeft$, $BorderRight$ are the boundary val-

ues of this component, $BorderTop, BorderBottom$ are the minimum row index and the maximum row index, and $BorderLeft, BorderRight$ are the minimum column index and maximum column index of it. The $\sum BinaryMap[BorderTop + i, BorderLeft + i : BorderRight - i]$ stands for summing up all pixels from column $BorderLeft + i$ to column $BorderRight - i$ in row $BorderTop + i$. If the amount is zero, column $BorderRight - i$ is hollow and is considered a gap.



(a) Clean block (b) Binary map of clean block

(c) Block containing image (d) Binary map of block containing image

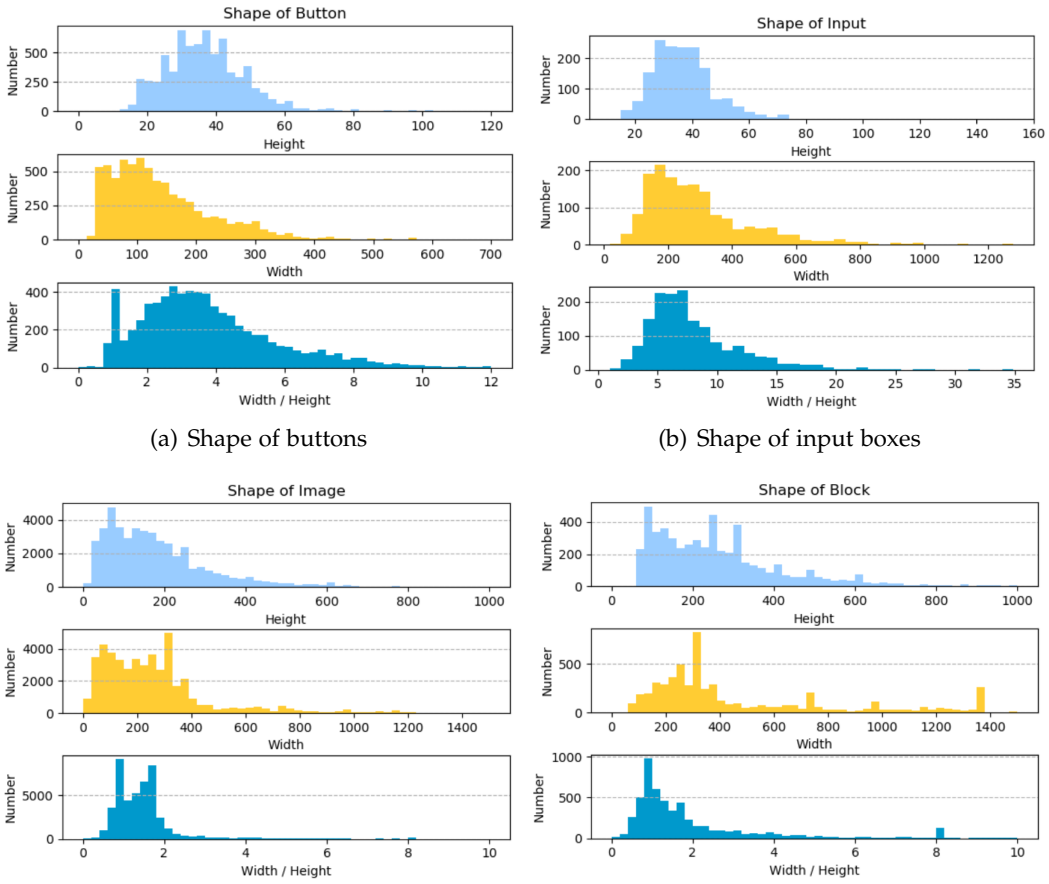
Figure 5.9: The demonstrations of a block. Blocks are drawn with green bounding box, and they usually are bordered regions where contain multiple components.

5.3.5 Irregular Shaped Components Selection

On the ground of the statistics 5.8, I observe the rule that human-computer interface components always have regular shapes (rectangle or round or oval), and the irregular objects are more likely to be image elements. However, the exception still exists in functional UI components, although it is relatively rare. Thus, I add this step to

check all irregular objects and estimate whether they should be selected as potential UI components or be filtered out based on heuristics of size and aspect ratio of elements.

According to the datasets collected from web set and mobile applications whose statistics is shown in Figure 5.10, some rules of the interface design in terms of the scale and length-width ratio are exposed and can be used as heuristic knowledge.



(a) Shape of buttons

(b) Shape of input boxes

(c) Shape of images

(d) Shape of blocks

Figure 5.10: Statistics of components' shape. For each type of UI components, three kinds of information are collected: height, width and aspect ratio (width / height). The amount of *Buttons*, *InputBoxes*, *Images*, *Blocks* are 10566, 3460, 39998, 1568 respectively from totally three different web and mobile application datasets.

From the above statistics, we can see the distribution of the shapes for different components. All of those distributions are long-tail, and the majority of data concentrate on specific ranges. For example, most of the buttons' height is in the range of 20 pixels to 60 pixels, and most of their width is between 40 pixels and 300 pixels, the major aspect ratio of buttons is on a scale of one to eight.

Therefore, adhoc filters are built according to the heuristics 5.1. Four rules are defined here, and the filters are scattered over the whole process when implemented.

The *SmallComponent* rule leaves out those small noises; the *AbnormalAspectRatio* rule is checked for all the components to filter out those impossible to be UI elements; for all irregular objects, the *IrregularImage* estimation is conducted to decide if we can directly affirm the irregular component is image; and the *Block* judgement should be pass for all blocks.

Number	Name	Heuristics
1	Small Component	$Area < 175 \bigwedge Perimeter < 70$
2	Abnormal Aspect Ratio	$width/height < 0.4 \bigvee width/height > 20$
3	Irregular Image	$isIrregular \bigwedge height > 70$
4	Block	$isBlock \bigwedge width > 70 \bigwedge height > 70$

Table 5.1: Heuristics based on the statistics.

5.3.6 Nested Components Detection

In the previous stages, we do not inspect into components to exam their contents, but some elements, such as button and input box, might be superimposed on an image. Therefore, the images detected are required to be further processed to check whether there are other interface components on them.

An observation about such nested components indicates that it is impossible to view a button inside another button or an input box nested in another input box. Also, technically, it is in no sense in putting a button inside another button; this behaviour is not allowed by the front-end language either. On the other hand, the common scenario is some interactive elements are put upon an image background, as shown in Figure 5.11.

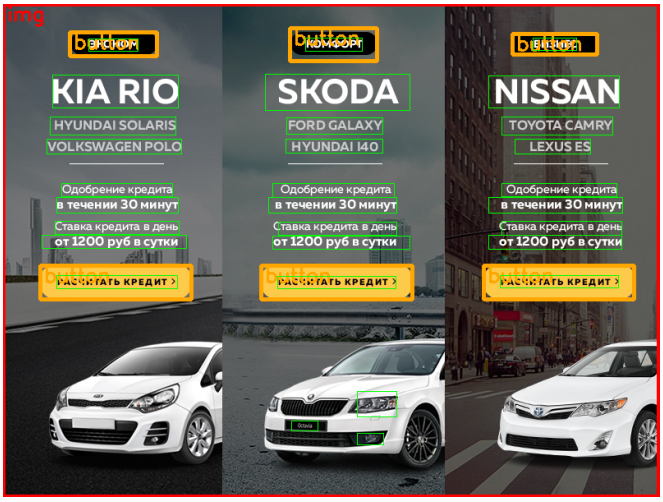


Figure 5.11: Example of figure that some interactive elements on a complicated image background

As elaborated in the pre-processing section 5.2, in order to overlook the detailed

texture and contents in the given image, I adopt a sensitive gradient threshold to try to incorporate all adjacent foreground points into individual connected components. But the drawback of this method is that the possible interface elements on an image are also integrated into this image component and be treated as a part of it.

To conquer this problem and separate the nested elements, a novel trick is proposed. I observe that the functional elements usually have a solid monochromatic background, which makes the gradient of their background zero. So I reverse the binary map to generate an opposite image, which means all the background points whose pixel values are zero are assigned value 255, and vice versa, all white points are transformed into black.

(a) Binary map (b) Opposite map

Figure 5.12: The demonstration of the binary map and its opposite image.

Figure 5.12 shows the binary map that integrates some buttons into the background and identifies them as an entire image component. But we can observe from the binary map 5.12(a) that a black hole in the shape of a button appears surrounded by white foreground points. While this binary map is reversed, all the background became foreground, and the black holes turn to white components. By this means, the nested elements are extracted and transferred to the beginning of the UI components detection pipeline to conduct the same processes again.

As the aforementioned observation, this task cares more about the situations that interactive elements superimpose upon background images. Thus, when selecting the potential components inside images, I only leave those rectangular objects that are more likely to be interface elements rather than contents of the image.

5.4 Classification

After all the possible human-computer components are picked up in the previous steps, a classifier is built to identify their classes in order to generate the proper code at the very end of this system. Prior to any technical details, the categories and classes of interface elements in this task are defined as table 5.2.

5.4.1 Categories and Classes of UI Components

The ultimate purpose of the UI components detection pipeline is to segment the potential interface elements from the input image and, based on their expected features, label them with human-computer interface tags, such as <img> or <button> in HTML, for code generation. Therefore, I define three categories of the UI components on the ground of the functional attributes those elements should have in the real interface, and I define six classes according to the related objects and tags that perform particular functionalities in the front-end languages.

Category	Class Name
Interactive Elements	Button
	Input Box
Static Resource	Image
	Icon
	Text
Layout Structure	Block

Table 5.2: The categories and classes of UI components.

Interactive Elements: Those who are explicitly associated with some actions, such as page jumps and close the window, and can gather instructions from users are grouped in this category. For instance, the buttons are always related to some functions that will be performed after clicking, and the input boxes are the places where the user can feed their information into the application.

Static Resource: This category indicates that the elements in this group focus on displaying contents instead of interaction with the user, just as the images and text on a webpage. Icon here is specifically useful for mobile application, it can be regarded as a tiny image, but it exists independently in Android development, so I separate this class from the image. Although those resources can sometimes be implemented as functional elements, typically as hyperlinks, I still treat them as a static resource at this stage for convenience. But at the code generation stage, I will give them the chance to expand their functions if need be.

Layout Structure: A special class distinguished from previous individual elements is Block, it is a layout structure that could contain multiple other kinds of elements. In other words, it presents a section of graphical interface consisting of various components. The meaning of this category is to find out some obvious hierarchies and layers for future code generation.

Figure 5.13 demonstrate the visualized examples of the human-computer interface components' classes in the real application. This is also a glimpse of the final result of the UI components detection pipeline that the input image is semantically

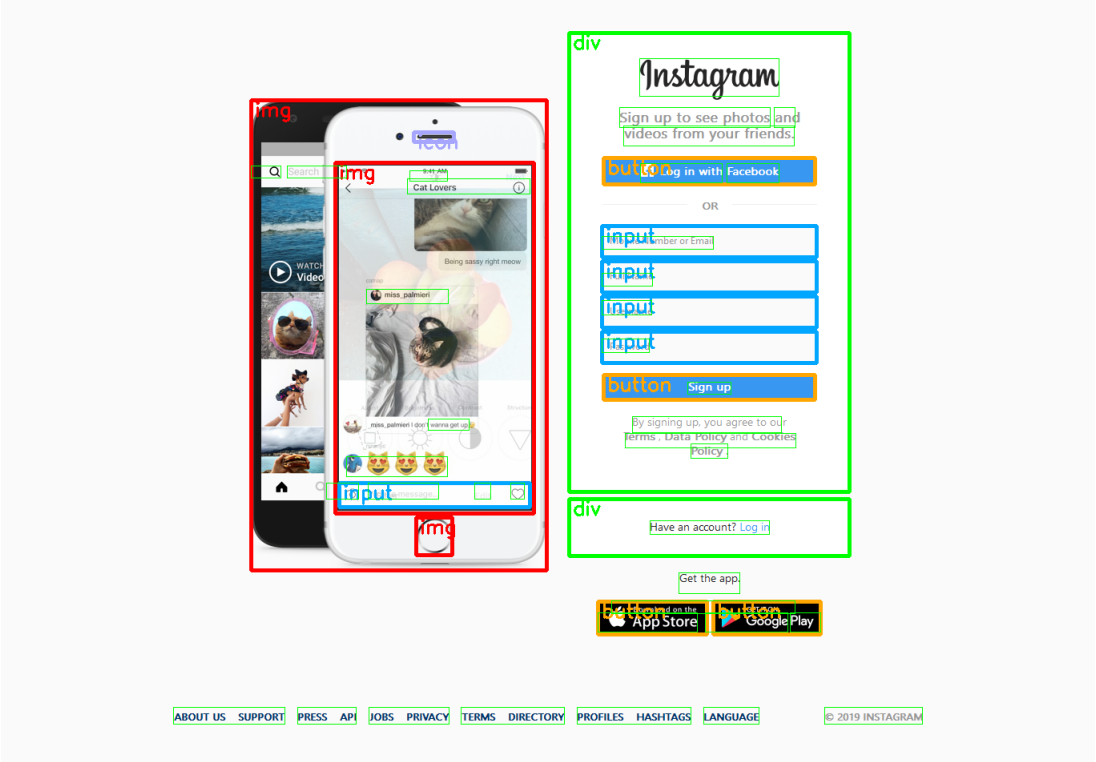


Figure 5.13: Demo of a labeled web page screenshot. Various classes are tagged with different colours of bounding box; the slim green boxes in this picture are the results from the CTPN showing the text recognition.

segmented into components with tags in which we know the locations and classes of those detected objects.

5.4.2 Classifier Model

To perform components classification, I recur to machine learning methods. Several techniques have been tried in the process including Super Vector Machine and Neural Network, and finally, a simple four-layer convolutional neural network is adopted for its best performance.

Since the emphasis of this part is not on the neural network, I only simply introduce the model's structure and evaluate the performance of various classifiers to sustain the choice of CNN.

There are three different methods implemented in this section, the Scale-invariant Feature Transform (SIFT) [Lowe, 2004, 1999], the Histogram of Oriented Gradients (HOG) [McConnell, 1986; Freeman and Roth, 1994] combined with Support Vector Machine (SVM) [Cortes and Vapnik, 1995] and the Convolutional Neural Network (CNN). Again, this part does not elaborate on the technical details of those techniques; instead, it just introduces the basic theories of them and states the experi-

mental results.

5.4.2.1 HOG + SVM

Another popular feature extraction technique is the Histogram of Oriented Gradients (HOG). It is usually be utilized as a feature descriptor in computer vision and image processing [Freeman and Roth, 1994]. This pipeline has some unique advantages compared to others. First, HOG processes image on a dense grid of uniformly spaced cells, which endows it the favourable ability to keep the optical and geometric invariant of the image. Besides, this algorithm is more robust and can be less sensitive for small changes [Dalal and Triggs, 2005].

This pipeline consists of five steps: gradient computation, orientation binning partition, descriptor blocks generation, block normalization and SVM object recognition.

Gradient computation: The first pre-processing step is similar to the UI components detection pipeline, where the gradient of this image is calculated at the beginning. Basically, the way to compute the gradient is the same as formulae 5.3, but the author implements it by the following filter kernel:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.4}$$

$$g(x,y) = w * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(x,t) f(x-s, y-t) \tag{5.5}$$

The kernel 5.4 is convoluted on image to compute the gradinet, the general expression of a convolution is stated in 5.5, where the $g(x,y)$ is the filtered image, $f(x,y)$ is the original input image and w is the kernel. Every element of the kernel is in range where $-a \leq s \leq a$ and $-b \leq t \leq b$.

Orientation binning: This step create cell histograms for the gradient result. The cells can be either rectangular of radial. In the radial shape, for example, the degree of each bin depends on the total number of channels, the degree of a bin in a night-channel histogram is $360/9 = 40^\circ$. The gradients cast a weighted vote for those bins, for instance, if a pixel's gradient direction is 12°, then the first bin (range from 0° to 40°) increases by x, where x is the magnitude of the gradient.

Descriptor blocks generation: The gradients vary greatly because of the illumination and contrast, so the strength of gradients should be normalized to acquire accurate result. To this end, the cells are grouped into larger spatially connected blocks, and those blocks are concatenated to a descriptor.

Block normalization: As mentioned in the previous paragraph, HOG conduct the

normalization to mitigate the huge variation among gradients' lengths. I adopted the L2-norm in my implementation; the expression of it is shown below:

$$f = \frac{v}{\sqrt{\|v\|_2^2 + e^2}} \tag{5.6}$$

where: v is the vector, $\|v\|_k$ is the k-norm of v which can be 1 or 2 to show L1-norm or L2-norm, and e is a small constant.

Object recognition: The previous steps generate a normalized feature vector of an image or object, then this vector can be feed into some classifiers to recognize its class. A widely used technique to combine with the HOG is the Supper Vector Machine (SVM) [Dalal and Triggs, 2005], a supervised learning model based on the learning algorithm to perform classification and regression [Patel, 2017].

The basic idea of SVM is that we try to separate p-dimensional vectors with $(p-1)$-dimensional hyperplanes. There can be multiple hyperplanes that might partition the data. The one representing the largest separation or margin between the two classes is selected as the maximum margin hyperplane. The distance from it to the nearest data point on each class is maximized [Asa Ben-Hur, 2001].

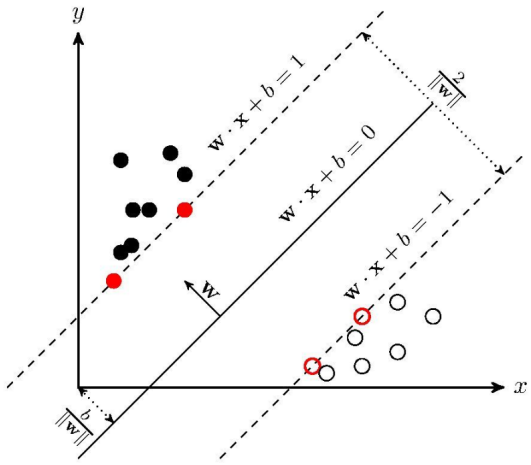


Figure 5.14: Hyperline partitioning two groups of points

Figure 5.14 [Yu, 2019] presents the demonstration of the linear SVM, where w is the normal vector and b is a constant parameter; the parameter $\frac{2}{\|\vec{w}\|}$ defines the offset of the hyperplane from the origin along the normal vector $\vec{w}$. All possible hyperplanes should satisfy the expression $\vec{w} \cdot \vec{x} + b = 0$ for the set of points $\vec{x}$. In this case, the $\vec{x}$ is the set of HOG feature vectors computed in the preceding steps.

5.4.2.2 SIFT

The Scale-invariant Feature Transform is a classic feature detection method widely used in computer vision tasks to describe the local features in images. It is adopted

in many applications such as object recognition and action recognition [Lowe, 1999], the robust performance in recognition tasks is the motivation that I try this technique.

As its name suggests, the core concern this algorithm addresses is to keep the features of objects invariant in various scales [Mikolajczyk and Schmid, 2005]. It searches for the extreme points in the space and extracts their location, scale and orientation, and then all those features are stored in the database through the Hash Table. An object from a new image is recognized by comparing and matching its features to the database to find the candidates based on Euclidean distance of their feature vectors. Mainly four steps involved in the SIFT algorithm, this section is just a brief summary of it [Mordvintsev and Revision, 2013].

Scale-space extrema detection: This step extracts a large set of feature vectors from the input image. Those vectors are theoretically invariant to image scaling and rotation and robust to local geometric distortion. To this end, the author proposed the scale-space extrema detection by using Gaussian filter in various σ scales, where the σ decides the smoothness of an image. Small σ reflects the details of the image, while large σ presents the blurred picture.

SIFT uses Difference of Gaussians (DoG) to calculate the d Gaussian Pyramid. DoG is obtained as the difference between two images blurred by different Gaussian filters with different σ, and this process is done for all layers in the Gaussian Pyramid as illustrated below.

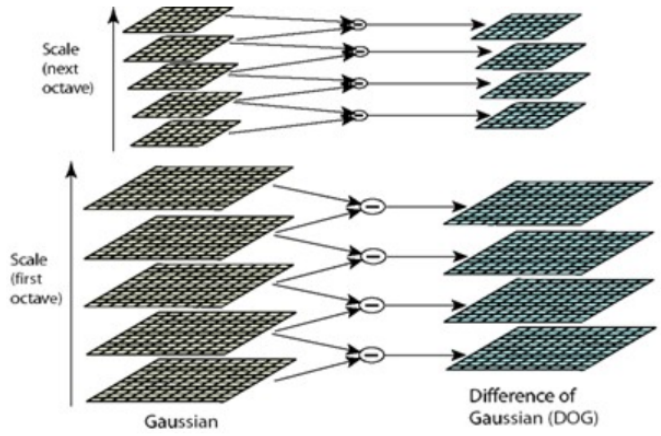


Figure 5.15: DoG are computed in all layers in Gaussian Pyramid

After the DoG is calculated, each pixel in an image is compared with its eight neighbours as well as corresponding nine pixels in the next scale and the previous scale to find the local extrema that can be a potentially key point representing the feature in its scale.

Keypoint localization: For all potential keypoints, SIFT uses Taylor series expansion of scale-space to get a more accurate location, but if the intensity at a point is less than a specific threshold, it is rejected.

Orientation assignment: The key points are assigned orientation in this step by calculating the gradients with its neighbours on this scale. The gradients' direction and magnitude are stored in an orientation histogram with thirty-six bins. This process is helpful to keep invariance to image rotation.

Keypoint descriptor: SIFT store the keypoints in a descriptor which consists of multiple bins of orientation histogram. In detail, a 16x16 neighbourhood around each keypoint is taken and divided into 16 4x4 sub-blocks. Then an eight-bin histogram is created for each sub-block to collect the gradients. So a total of 128 bins are created and vectorized to store the information of this keypoint.

Eventually, the features of an image are extracted and can be used to recognize others in the same category either by matching the Euclidean distance or by utilizing this vector as an encoder and feed into machine learning techniques such as SVM.

5.4.2.3 CNN

Compared with the aforementioned techniques, the Convolutional Neural Network (CNN) is a more end-to-end method [Jeeva, 2018]. Because of the popularity of CNN, I will not iterate its basic mechanism and principle in this thesis. The emphasis here is on the model's structure and its effectiveness.

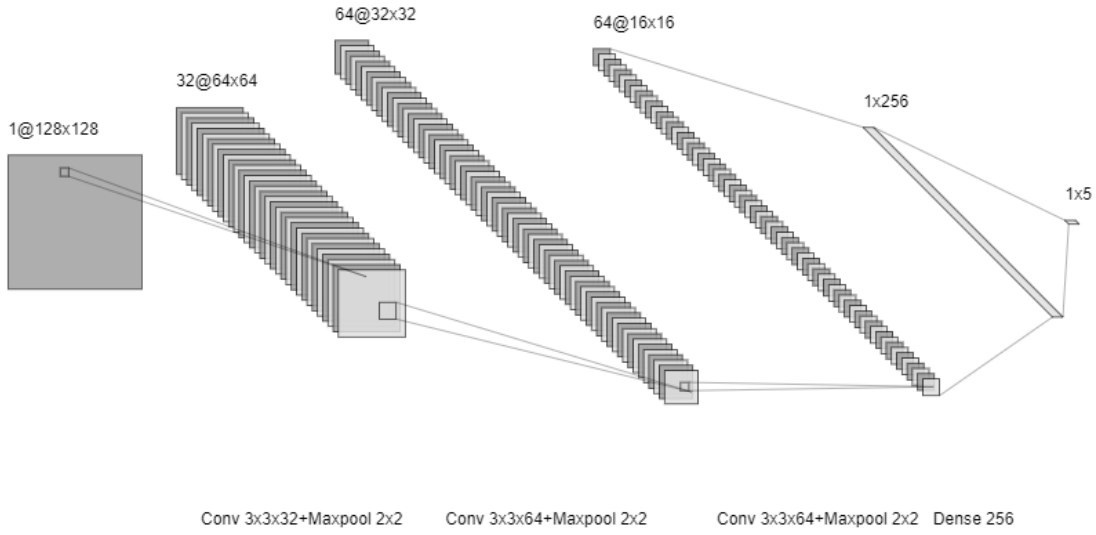


Figure 5.16: The structure of the four-layer network. A 3x3 sliding window is adopted to move through the original image that is resized into size of 128x128. All convolutional layers are followed by a 2x2 max-pooling layer. The output layer is a five-class softmax to classify the input into *image*, *button*, *input box*, *icon* and *text*.

As the classic use case, this technique is utilized to recognize the class of image in the UI components detection pipeline. The input now is the individual component.

An observation on the datasets of human-computer interface components is that the buttons and input boxes are less variants. In other words, components in those two classes are always in a similar pattern. For example, the buttons are always in the form that one or few words located in the centre of a bordered region. On the contrary, the image elements are always variegated and colourful. So the difference among various classes can be rather obvious, and a shallow neural network might be sufficient in this case.

Therefore, I build a four-layer model to handle this task, and the structure is presented in Figure 5.16. In order to offset the effect caused by the insufficient parameters in a shallow network, each layer is created more broad.

5.4.3 Performance

As mentioned above, I implement and compare those three techniques to select the best one as the classifier. The experiment is conducted on the components dataset consisting of images of web and mobile application. Although there are some slight variations among interface elements of web and mobile apps, those nuances do not impose considerable influence on the model's performance. Therefore, all components in the same class from different sources (web and mobile application) are collected together and fed into the training models. The table 5.3 shows the experimental performance of those three classifiers.

Model	Accuracy	Recall	Balanced Accuracy
SIFT + SVM	0.9166	0.9061	0.9007
HOG + SVM	0.9326	0.9112	0.9065
CNN	0.9566	0.9128	0.9226

Table 5.3: The performance of models. The balanced accuracy is taken into account as a criterion because of the imbalanced datasets where image components are far more than others. The experiments present that the CNN model is relatively better than the other two in all aspects.

5.5 Text Processing

The text processing is achieved by a popular text detection method, Connectionist Text Proposal Network (CTPN) [Tian et al., 2016]. The CTPN was originally designed for accurately localizing the text lines in a nature scene, leveraging the vertical anchor regression and connectist proposals to detect text lines accurately. This technique achieves high performance on the ICDAR 2013 and 2015 benchmarks at a fast processing speed (0.14s/image).

5.5.1 Introduction

The CTPN refers to the state-of-the-art object detection methods, especially the Faster RCNN [Ren et al., 2015], but the authors proposed some novel improvement to adapt the network to the natural scene text line detection based on the visual properties of the sentence. In this process, several following contributions are made:

First, the authors transformed the OCR problem into localizing a sequence of fine-scale text proposals, which could apply some mechanisms of object detection means [Girshick, 2015; Ren et al., 2015]. For instance, anchor regression that widely used in recent object detection methods are utilized to acquire the position of the targets. But CTPN takes the morphological characters of text line into consideration and deploys vertical anchors to produce the region proposals, which makes significant progress to the localization accuracy. This novelty departs from the Region Proposal Network in the Faster RCNN, which attempts to predict a whole object, hence is difficult to provide a satisfying localization accuracy in the case of text detection.

Second, in view of the consecutiveness of the sentence, the authors incorporated an in-network recurrence mechanism that connects sequential proposals in the convolutional feature maps into the model. Thereby, the network is capable of exploring meaningful context information.

Third, this method is scalable and robust in various environments. An end-to-end trainable network is produced, able to handle multi-scale and multi-lingual text in the same image without and revision and filtering. This robustness is achieved by the diversity of the training dataset as well as the anchor regression mechanism.

Attribute to the aforementioned novelties, the CTPN refreshed a series of benchmarks, significantly improving results of preceding methods. (e.g., 0.88 F-measure over 0.83 in [Gupta et al., 2016] on the ICDAR 2013, and 0.61 F-measure over 0.54 in [Zhang et al., 2016] on the ICDAR2015).

5.5.2 Technical Details

Three critical novelties are proposed in the CTPN: detecting in fine-scale proposals, recurrent connectionist text proposals and side-refinement. Figure 5.17 shows the architecture of the CTPN.

Fine-scale proposals: This technique adopts the fully convolutional network to allows an input image of arbitrary size. Referring to FRCNN, it leverages slide windows to detect the text areas in the convolutional feature maps, then generates a series of fine-scale text proposals.

The sliding-windows methods used to facilitate the region proposal. Most classical sliding-windows approaches define several fixed-size anchors to detect objects of similar size. The CTPN extends this efficient mechanism by means of revising the scale and aspect ratio of anchors to fit the properties of text. One peculiarity of a text line is that it is a sequential region where it does not have an obviously closed boundary. Multi-level components, such as stroke, character, word, text line and paragraph, are involved in the process. The text detection in this case, however,

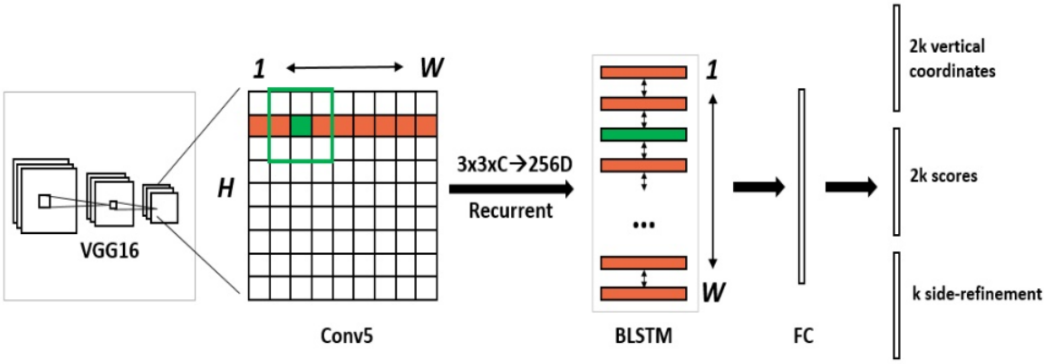


Figure 5.17: The overall structure of the Connectionist Text Proposal Network (CTPN). A 3x3 sliding window is applied through the last convolutional map of the base network (VGG16). Then the sequence of windows in each row is recurrently connected by a Bio-directional LSTM to gather the sequential context information. In the end, the RNN layer is connected to a 512D fully-connected layer and the output layer where the text/non-text score and y-coordinate are predicted, and the k anchors are offset by the side-refinement.

focuses on the text line and region level, so the accuracy might not be satisfied if it still treats the targets as single objects.

Therefore, the authors proposed a vertical mechanism that predicts a text/non-text score and the y-axis location of each proposal. The experiments prove that detecting text line in a sequence of fix-width proposals is more effective and accurate than recognizing individual characters. Moreover, the fixed-width proposals also work well the text of various scales and aspect ratios.

In detail, the CTPN utilizes k vertical anchors in a fixed width of 16 pixels. Those k anchors have the same horizontal location, but they vary in k different heights vertically. Thus, each predicted proposal has a bounding box with a size of h x 16, where the h is the predicted height. The network's output is the text/non-text scores and the predicted y-coordinates that represent the height of this anchor for k anchors of each window.

Recurrent Connectioist Text Proposals: The fine-scale proposals result from splitting the text line into a sequence of slender regions, and those proposals are predicted independently. However, it is obviously not robust to process each part of text regions separately. Therefore, the authors leveraged the recurrent mechanism to make use of the sequential nature of the text. Furthermore, the CTPN integrates the context information into the convolutional layer.

To this end, the CTPN, the long short-term memory (LSTM) [Hochreiter and Schmidhuber, 1997] is used for the recurrent neural network (RNN) layer. The authors adopted some tricks to address the vanishing gradient problem. Moreover, they use a bi-directional LSTM [Graves and Schmidhuber, 2005] to extend the RNN layer to collect the context information in both directions.

Side-Refinement: After acquiring the fine-scale proposals, the CTPN straightforward connects those whose text/non-text score is greater than 0.7 to construct the text line. The text regions are now divided by a sequence of equal 16-pixel width proposals of different heights. Such variations can lead to inaccuracy of text line detection. To address this problem, the authors proposed a side-refinement approach to estimate the offset for each proposal horizontally.

5.6 Merge

The UI components detection and text detection are done independently by the two branches. In the end, the pipeline cross-checks the correctness and integrates those results.

One drawback of the image processing based method is that the noises would be selected by mistake. The pre-processing stage attempts to incorporate all the variegated contents into individual components and plenty of objects that are unlikely to be interface components are filtered out based on their sizes and aspect ratios according to the heuristics in table 5.1. But there are still some isolated regions where are wrongly selected as component candidates. The observation on those false positives is that they always come from two sources, the small isolated parts of the image components and the text regions.

As stated at the beginning of this chapter, the detection of interface components and recognition of text regions are separate and performed in two specified pipelines. Those two branches are expected to focus on their won tasks for the sake of accuracy. Therefore, the text regions are not desired in the UI components detection pipeline and should be filtered out as noises. To this end, the CNN in this branch is also trained to be able to recognize the text to avoid mixing it up with other interface elements.

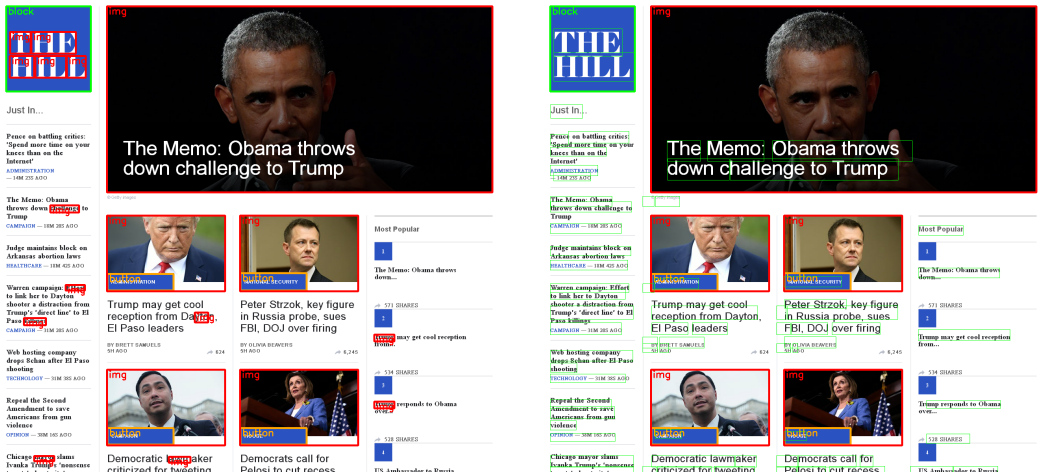
But the real text regions still have chance to be improperly recognized as buttons or images because of the false prediction of the CNN. Thus the system applies the results of CTPN to double-check the results of components detection and discard those misidentified human-interface elements. The process is visualized in Figure 5.18.

To achieve this, the resulting UI components on the left-hand side are filtered by computing the overlap with the text lines. The overlap computation is on the ground of their *Intersection over Area* (IoA), as $IoA(a,b) = \frac{Area_a \cap Area_b}{Area_a}$. A valid UI component i should satisfy the two requirements:

$$\forall j \in Text(IoA(i,j) < 0.7) \tag{5.7}$$

$$(\sum_{j \in Text} Inter(j,i))/Area_i < 0.85 \tag{5.8}$$

The requirement 5.7 means the intersection area of a single text region with the component i should be smaller than the 70 percentage of component i's area; and the



(a) Result of the UI components detection (b) Result after merging with text detection

Figure 5.18: A section of web application interface. The 5.18(a) demonstrates the detecting result of the UI components detection pipeline, in which several text regions are wrongly recognized as image elements (marked with red bounding boxes). The merged result is shown in the 5.18(b), the green slim lines in this figure are the text areas detected by the CTPN. After double-checking by the CTPN, those false positive image elements that are actually text are discarded.

expression 5.8 stipulates that the total text area of an interface element should less than 85 percentage of its area.

Eventually, the merged result is presented to the user as the visualized output of this whole detection system, and it is also transferred to the code generation section to produce the corresponding front-end code.

5.7 Summary

This chapter presents the UI components detector of UI2CODE in which two branches are involved: graphical component detection and text recognition. I built a pipeline utilizing image processing algorithms to achieve the graphical component detector. Three steps compose this part: (1) pre-processing that manufactures the binary map by calculating gradients of the input image; (2) components detection section segmenting the connected components and select the potential UI element candidates; (3) a CNN classifier to categorize the components into several predefined classes. Meanwhile, I utilized a powerful text recognition model CTPN to detect the text areas of the UI. In the end, the results of two branches are integrated to produce the final result.

Chapter 6

Code Generation

Code generation module, functioning as the output section, produces the front-end code that implements the input UI design. This part combines components detection results from the previous step and some UI layout structure identification algorithms based on common practices of UI development to generate the deliverable.

Through the processing of the above pipeline, we acquire the spacial positions and classes of UI widgets. To facilitate developers and designer more straightforwardly, as mentioned in Chapter 1, UI2CODE converts the detected information into usable front-end code (HTML, CSS). However, there are still some gaps between the detection result and the professional program. For example, the output locations of elements are absolute coordinators on the image. It is not sufficient for front-end programming where the position of a component is always relative to its parent container and neighbours[WHATWG, 2019], such as 50 pixels from the left element. In addition, the proper hierarchy among UI components is also critical for a maintainable program Harris [2014].

Therefore, I proposed several algorithms and cascaded them in a pipeline to produce the final deliverable code. In order to bridge the gaps between detection and deliverable program, this technique includes two high-level parts: (1) hierarchical segmentation and (2) front-end code generation. The hierarchical segmentation adopted some image processing algorithms to detect cutting lines which always be partitions of blocks on a UI and applied some strategies to decide the hierarchy of regions. The front-end code generation phase only focuses on HTML and CSS languages in this stage. We investigated practices of web development to produce a high-quality code that is easy to expand and maintain.

To evaluate the effectiveness of this technique, I ran the generated code on the browser and compared its visual effect with the input UI design [White et al., 2019]. However, due to a lack of time, this part is rather not robust and requires more improvement in terms of methodology and evaluation. More user study and benchmark is needed to judge the quality of the created code.

This part of the work has not been completely finished yet. Hence, this chapter only introduces some of the progress and outcomes so far, including a cutting line detection algorithm based on image processing, block segmentation, hierarchy establishment, and web code generation referring to standards and usual practices of web development.

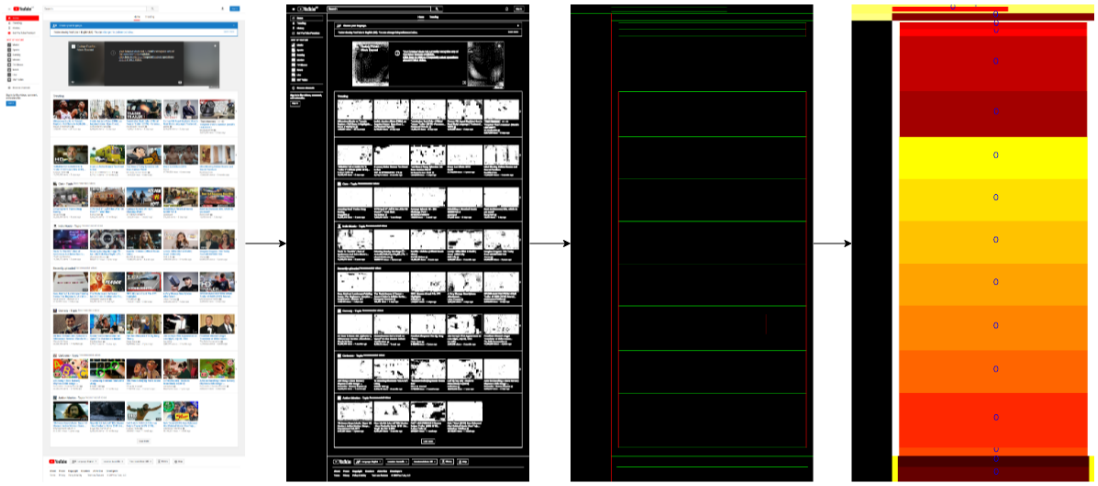


Figure 6.1: Visualized demonstration of hierarchical block segmentation. From left to right are: (1) the input or a web UI which is a full-size screenshot from YouTube; (2) the binarized gradients map of the original image; (3) the result of cutting line detection, where we only care horizontal lines and vertical lines; (4) the hierarchical blocks with various colours segmented based on the cutting lines.

6.1 Hierarchical Block Segmentation

In order to reconstruct the layout of a UI, we need first segment it into several regions, or blocks we call them here. Typically, all UI elements can be categorized into two class, *block level elements* and *inline elements* [Tutorialspoint, 2016]. Each block can be regarded as a container where single or multiple UI elements are gathered in it, and it can be the fundamental layout structure of a UI [MDN, 2019]. Therefore I proposed a technique, in accordance with solid borders of each container combined with gaps between components, that divides an entire image into various blocks as sub-regions of the UI and establishes hierarchies among them.

Blocks we acquire from the previous steps in section 5.3.4 can be merged with the result of this pipeline at the end, but we cannot only count on that approach for several reasons. First, the previous block recognition method requires blocks having an entire rectangular boundary comprising four borders. However, sometimes a block in UI only has one or two or even no borders, such as a single upper border or a left border; in that case, the algorithm is no longer available. Second, that algorithm is designed to apply in extracted components, which is incompatible with the situation that we want to segment the regions rather than analyze all elements directly.

Thus, I proposed a new approach to settle down the problems. The technique is composed of several steps: (1) detecting the explicit partitions of blocks, which are always their boundary lines; (2) segmenting the UI into blocks on the basis of those cutting lines, as well as some implicit information between elements such as gaps;

(3) establishing their hierarchy for further code generation.

Figure 6.1 presents the visualized pipeline in which we take the UI design in any size as input, and convert it into the binary map by applying the pre-processing proposed in the last chapter 5.2. Then I built an efficient line detection algorithm to acquire cutting lines or partitions, and these lines are considered as solid borders of blocks. The blocks are segmented according to the lines, and their hierarchies are decided according to their relative positions and inclusion relations.

6.1.1 Cutting Line Detection

As mentioned above, the purpose of this step is to detect partition lines of blocks, which are also their boundary lines. To this end, this section introduces an image processing based method to analyze the UI design. However, unlike the existing popular line detection algorithm, such as Hough Transform [Duda and Hart, 1972], here we do not need to concern complicated and universal situations; instead, we only focus on particular cases based on the characteristics of UI.

One observation on web UIs is that borders of elements implemented by HTML or CSS are always rectangular [W3Cschool], and they are either horizontal or vertical, as shown in Figure 6.2. Therefore, the target lines we desire to find are also should be these two types, which requires a horizontal and vertical line detection algorithm.

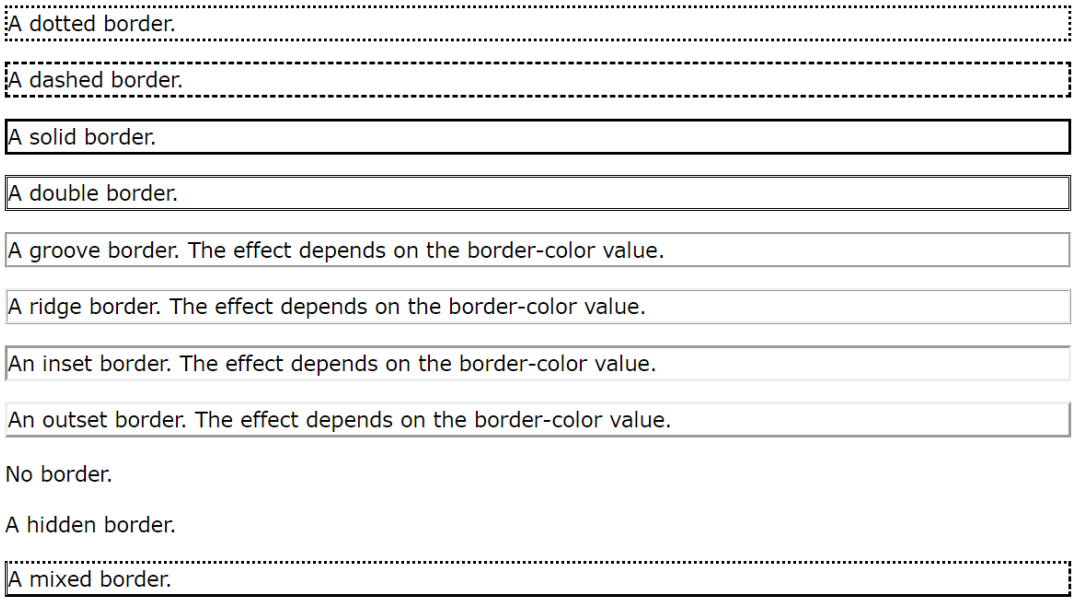


Figure 6.2: Various borders of HTML elements, but the common characteristic is that they are all rectangular.

Thus, I proposed an efficient method to fulfil this need. This approach refers to the block recognition algorithm in 5.3.4. One common assumption with that is there should be a gap between the border of a block and the nested object within it. So,

this method identifies a point as part of a line by inspecting its neighbouring pixels. For instance, the points above or below a horizontal line should be background (zero pixel); similarly, the points next to the left side or right side of a vertical line should be background.

As the graphical components detection pipeline, the process also feeds on the binary map of the original image where the foreground is clearly exposed as white points. Then we scan row by row to find horizontal lines and column by column to acquire vertical lines. The pseudocode is presented in algorithm 5.

Algorithm 5 Line Detection - Horizontal

Input: Binary map, Minimum gap between border and contents, Minimum length of line

Output: An array of lines, each lines contains a start point and an end point

```
1:  HorizontalLines ← [ ]
2:  x ← 0
3:  for x in range(Row) do
4:      FoundLine ← False
5:      for j in range(Column) do
6:          if Map[x][j] = 255 and FoundLine = False then
7:              Head ← j
8:              FoundLine ← True                          ▷ Start a new line
9:          else if (Map[x][j] = 0 or j = Column − 1) and FoundLine = True then
10:             End ← j
11:             FoundLine ← False                         ▷ End the line
12:             if Head − End > MinLength then            ▷ Check the length and gap
13:                 if ∑_{i=x+1}^{x+Gap} ∑_{j=Head}^{End} Map_ij = 0 and ∑_{i=x−1}^{x−Gap} ∑_{j=Head}^{End} Map_ij = 0 then
14:                     HorizontalLines.append((x, Head), (x, End))
15:                 end if
16:             end if
17:         end if
18:     end for
19: end for
20: return HorizontalLines
```

This algorithm shows that the process of detecting horizontal lines. The method for vertical lines is rather similar. We scan the binary map row by row and detect the foreground where the pixel value is 255, and each line is terminated when encountering a background point or the margin of the map. Then we check the length of these lines and inspect its upper and lower neighbouring rows to check gaps. While detecting vertical lines, we only need to change the row-by-row scanning to column-by-column scanning and inspect the left and right side neighbours rather than up and below ones. The algorithm is efficient due to its simple complexity, as well as sufficient for UIs because of their cutting line' s unique property.

6.1.2 Block Segmentation

After acquiring cutting lines, we segment the entire UI into independent blocks as the fundamental layout structure elements. The objective of this step is to enhance the usability and maintainability of the further generated code in the way that fits usual practices of web development [Musciano and Kennedy, 2007; WHATWG, 2019; Harris, 2014].

This problem is similar to the *Plane Division by Lines* problem [Engel, 1997; Graham et al., 1994]. It is a classical mathematical problem that questions the maximum number of regions divided by n lines in the plane. But here we care more about positions of these divided regions rather than their amount. In other words, the challenge in this case is how to split the entire image by multiple intersecting vertical and horizontal lines.

One characteristic in HTML elements is that the length of a border amounts to the length of its relevant side. For instance, the length of the left and right border of a block is exactly equal to the block's height, and the length of the block's upper or lower border is equal to its width. Therefore, given one line, we can divide the plane into two zones with a certain size. For example, in Figure 6.3, given $line_a$, the plane is split into $region_1$ and $region_2$ where widths are the same as the length of $line_a$. Similarly, widths of $region_3$ and $region_4$ are equal to the length of $line_b$.

This division already exposes some hints of region's hierarchy. $region_3$ and $region_4$ can be regarded as nested blocks of $region_2$, in other words, these two regions are children nodes of $region_2$ in HTML. According to the definition and attribute of child node in HTML, the children should always be completely contained by its parent [Lee, 2012]. Thus, the upper boundary of $region_3$ should lower than or equal to the position of $line_b$, and the lower boundary of $region_4$ should higher than or equal to the lower bound of $region_2$.

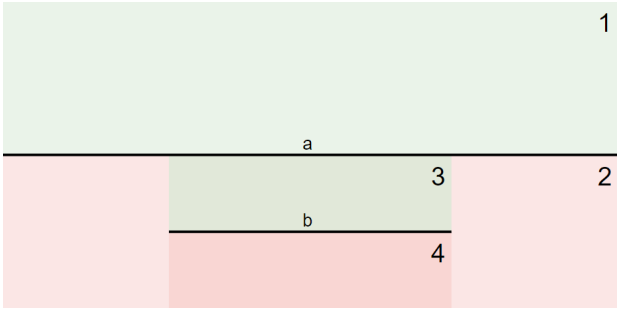


Figure 6.3: Two horizontal lines a and b divide the plane into four regions. Widths of all regions are equal to their cutting lines' length, while their heights are related to their parents.

Figure 6.3 shows the division of horizontal lines. Similarly, a vertical line splits plane into two zones where their heights are the same as the length of the line, and their width cannot exceed the left or right bound of the parent region. Figure 6.4(b) illustrates a more complicated situation where both vertical and horizontal lines are

involved. We give priority to the horizontal lines because block-level elements are always followed by a line-break which puts the next block below this one [Tutorialspoint, 2016]. Thus, if a block split by a vertical line is identical with anyone divided by a horizontal partition, we simply ignore it.

To this end, I proposed an algorithm taking the cutting lines as input and output block divisions. It consists of two parts: (1) each horizontal line is assigned an upper bound and a lower bound, and each vertical line is assigned a left bound and a right bound; (2) according to lines and their bounds, we generate the block defined by two points, the top-left and the bottom-right.

Algorithm 6 Block Division - Horizontal

Input: An array of cutting Lines comprising vertical and horizontal lines
Output: An array of blocks represented by top-left and bottom-right corners

$Upper \leftarrow NumberofLines \times [0]$ ▷ Initialize upper line for each line
$Lower \leftarrow NumberofLines \times [HeightofImage]$ ▷ Initialize lower line for each line
for i **in** $range(len(Lines))$ **do**
    for j **in** $range(len(Lines))$ **do**
        if $i \neq j$ **and** $Lines_i.Head_{column} < Lines_j.Head_{column}$
        and $Lines_i.End_{column} > Lines_j.End_{column}$ **then**
                ▷ Check inclusion relationship between two lines
            if $Lines_i.row > Lines_j.row > Upper_i$ **then**
                $Upper_i \leftarrow Lines_j.row$ ▷ Get closer upper line
            end if
            if $Lines_i.row < Lines_j.row < Lower_i$ **then**
                $Lower_i \leftarrow Lines_j.row$ ▷ Get closer lower line
            end if
        end if
    end for
end for
$Blocks \leftarrow []$ ▷ Generate Block
for i **in** $range(len(Lines))$ **do**
    if $Lower_i - Lines_i.row > MinHeight$ **then**
        $Blocks.append((Lines_i.Head), (Lower_i, Lines_i.End.column))$
    end if
end for
for i **in** $range(len(Lines))$ **do**
    if $Lines_i.row - Upper_i > MinHeight$
    and $(Upper_i, Lines_i.Head.column), (Lines_i.End)$ **not in** $Blocks$ **then**
        $Blocks.append((Upper_i, Lines_i.Head.column), (Lines_i.End))$
    end if
end for
return $Blocks$

Algorithm 6 presents the horizontal version of block division, which is fed on horizontal cutting lines and divides the image by them. The first step is to assign an

upper bound and a lower bound for each line. I defined an *Covering Relation* for a pair of lines in the way that horizontally, if $Line_a$'s start point's column coordinate is less than that of $Line_b$'s, and $Line_a$'s end point's column coordinate is larger than that of $Line_b$'s, then we say $Line_b$ is covered by $Line_a$. For example, in Figure 6.3, $Line_a$ covers $Line_b$. In this case, the upper bound of the upper $Block_3$ generated from $Line_b$ should be $Line_a$. Secondly, we generate blocks on the basis of the lines and their bounds. This process is conducted in both upper bounds and lower bounds, and it filters out the redundant blocks to get the final set of blocks.

6.1.3 Hierarchy Establishment

We now can establish the hierarchies among blocks. The fundamental principle here is that if a block is contained by another, then it is the child node of the container. This definition fits the node's property in HTML [Lee, 2012]. Therefore, this question can be converted to judging the inclusion relations of blocks.

The format of the presentation of a block is *(Top-left, Bottom-right)* which is a set of two points. So it is rather straightforward to judge the inclusion relation between two blocks: if $Block_a$'s top-left corner is in the upper left of $Block_b$'s top-left point, and the bottom-right corner of $Block_a$ is in the lower right of $Block_b$'s bottom-right corner, then we can say than $Block_a$ contains $Block_b$. However, this criterion only applies to a pair of blocks, while we have to draw the overall hierarchy of all blocks. Understandably, if we simply apply the above inclusion criterion, a block would turn out to have multiple containers. For example, in Figure 6.4(b), $Block_{14}$ is not only contained by $Block_6$ but also contained by $Block_4$. But we only want to acquire the immediate parent of a block [imm, 2019], such as parent $Block_6$ for children $Block_{14}$ and $Block_{15}$.

Algorithm 7 Block Hierarchy

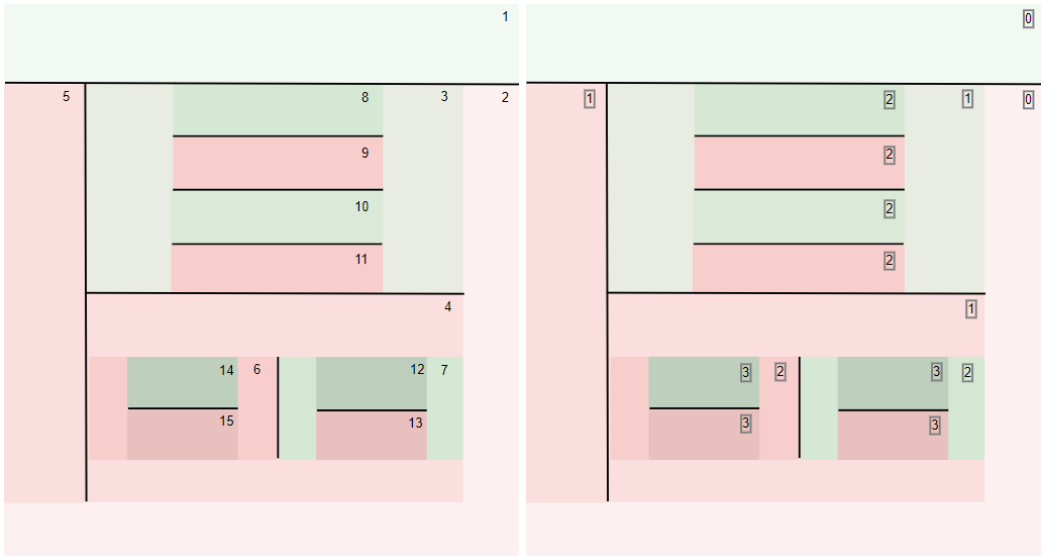
Input: An array of blocks represented by top-left and bottom-right corners
Output: Each block acquire its children and the immediate parent

```
for i in range(len(Blocks)) do
    for j in range(len(Blocks)) do
        if i ≠ j and Block_j contains Block_i then
            Block_i.Parents.append(Block_j)
        end if
    end for
end for
for i in range(len(Blocks)) do
    for P in Blocks_i.Parents do
        Blocks_i.Parents.remove(P.Parents ∩ Blocks_i.Parents)
    end for
    Blocks_i.Parents.Children.append(Blocks_i)        ▷ Only one immediate parent
end for
```

Therefore, I proposed a method shown in Algorithm 7 to establish the holistic

hierarchy. This algorithm first compares each block to all others to judge containing relations. Then, it follows a principle to pick the immediate parent that my parent's parent is not my parent. So, the block filters out all its parents that are also its parent's parents, and only one node will be selected at the end.

Now, we can travel from the root nodes that have no parent node through its children nodes to draw the global hierarchy. Figure 6.4 illustrates the division results and the hierarchical map generated based of the inclusion relations of blocks.



(a) Block division (b) Hierarchical map of blocks

Figure 6.4: Illustration of block division by cutting lines, and the hierarchy is established by checking the inclusion relations among these blocks.

So far, we've acquired the UI components information from the detection pipeline and the hierarchical layout structure from approaches above in this chapter, and we can produce the deliverable code based on these results.

6.2 Web Code Generation

This section yields the final deliverable of the UI2CODE, a set of front-end programming files including HTML and CSS, which implements the identical visual effect of the input UI design as well as some expected functionalities for certain UI widgets, such as button and input box. To this end, all the detection and division results are combined and converted into programming expressions. To be more specific, I transformed all the sizes, contents and spacial positions of UI elements into HTML code and reconstructed the layout of the design according to the divided blocks. Furthermore, in order to make enhance usability and maintainability of the generated code, I referred to some usual practices of real web development and discussed with some professionals to the feedback of the artificial program.

As mentioned in chapter 2, few similar works have been done in this field, but we can still have inspiration from some indirectly related works. For example, some of the web data extraction tools involved methodologies of formalization of web code. Several works [Gupta et al., 2003; Le et al., 2006; Cosulschi et al., 2006] provide a though that treats the web UI as a structured file and analyzed it systematically. I adopted some approaches, especially the Document Object Model (DOM), when producing the HTML code.

6.2.1 DOM Tree

In order to manufacture a high-quality program, I referred to the Document Object Model (DOM) specification. DOM is a set of platform and language-independent application programming interface that treats an XML or HTML document as a tree structure in which each node is an object representing a part of the document [Whitmer, 2009]. Therefore, the classified UI components from the previous detection pipeline are constructed as the leaf node of the DOM tree, and they are clustered into their container blocks which are roots and branch nodes [tre, 2019] in this tree. Then, the branches are used to represent the hierarchies among end elements and blocks.

The tree in Figure 6.5 illustrates the tree representing the hierarchical structure of the division in Figure 6.4. In HTML, we use <body> tag to define an HTML document [Musciano and Kennedy, 2007] where all contents of a web page are written in this area, so the root of the DOM tree is also <body> tag. The branch illustrates the relation between a parent node in a higher level with smaller layer number and a children node with bigger layer number.

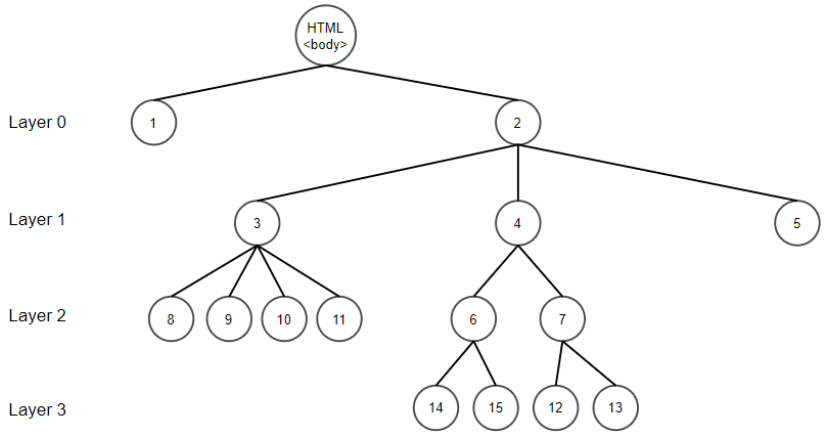


Figure 6.5: A tree constructed for the segmentation in Figure 6.4. The root node is the HTML <body> node which defines the document's body. Totally four layers are involved here to present the hierarchy.

The DOM tree is essentially a data structure for storing the information of an HTML file from which we can acquire the layout structure and the contents of each node of the web page. Thus, we now can fill in this tree with the detected information

of UI elements from previous steps to express a web UI. For example, Figure 6.6 illustrates a HTML DOM tree for a real-entire HTML code below.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Title</title>
</head>
<body>
    <div>
        <h1>Web page template</h1>
        <a>Click here</a>
    </div>
    <div>
        <div>
            <img src="image.png">
        </div>
        <div>
            <input>
            <button id="but1">Submit</button>
        </div>
    </div>
</body>
</html>
```

Here is an entire HTML code for a simple web page. It is worth noting that the <*div*> element actually functions as a container, it is used to cluster multiple end UI elements and create an independent layout block where it can be rendered separately [Musciano and Kennedy, 2007].

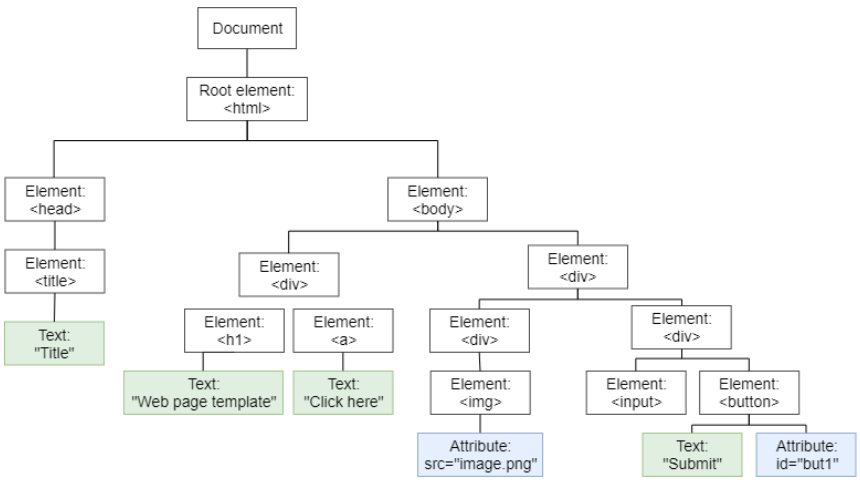


Figure 6.6: The HTML DOM tree for above web page code.

In the instantiated HTML DOM tree, the leaf nodes present attributes, such as id, class, href, src, and contents of end UI elements [Lee, 2012]; branch nodes are web

elements which can be rendered and filled contents; the root node is the <*html*> tag which defines the whole scope of this HTML file.

6.2.2 HTML Generation

This section presents an approach to implement the conceptual DOM tree into real code. It is a fairly straightforward process to generate code from the DOM, but several corners are worth noticing to produce a high-quality program in terms of usability and maintainability. However, this part is still immature yet, so I only introduce some principle and present the semi-manufactured product that needs further improvement.

The first issue is how to make the product easy to read and use. As mentioned in chapter 1, one obvious drawback of the downloaded source code of a real website is that there would be various customized element classes and attributes, which cause the difficulty to understand. To avoid the problem, I assign a unique and manageable ID to each element, so that the user can systematically control elements by referring their IDs.

Second, to enhance the extendability, I reverse some interfaces for functional UI widgets. For example, the *onclick* function is added to the <*button*> element's attribute for further function implementation.

Third, in order to maintain the layout structure in a convenient and decoupling way, the spatial positions of elements are always relative to their parent node and peer nodes. So I recalculate the coordinates of elements according to its container. Furthermore, as usual practice of web development, the rendering program is separate from the HTML layout and restored in CSS file.

```
<div id="div1">
    <div id="div3">
        <h1>Register Now!</h1>
        <p>Ready for joining the No.1 Jazz Club!</p>
    </div>
    <div id="div4">
        <input id="input1">
        <button id="but1" onclick="">Submit</button>
    </div>
</div>
<div id="div2">
    <div id="div5">
        <img id="img1" src="image1.png">
        <img id="img2" src="image2.png">
    </div>
</div>
```

Here we see a piece of generated code where each element is assigned a unique ID for the sake of future control. And the functional widget, such as the *but1*, has a interface *"onclick"* for implementing the real action of this button in future.

```
#div1{
    height: 200px;
    width: 400px;
}
#div2{
    height: 350px;
    width: 400px;
}
#img1{
    height: 150px;
    width: 300px;
    margin: 20px 30px 20px;
}
#img2{
    height: 150px;
    width: 300px;
    margin: 20px 30px 20px;
}
```

The above code is a section of the corresponding CSS rendering file in which implementing the size, location and other rendering attributes, such as background colour, as well as some special layout attribute, such as text-align and float.

6.3 Issues and Limitations

However, this part of the work is still immature. Several issues have not been addressed and require more in-depth researches. First, I only produced the CSS and HTML code of a web page, which do not include the real actions and functionality of elements. The functional part of a real website is usually implemented in Javascript [Musciano and Kennedy, 2007], so the future work should take Javascript part into consideration as well. Second, the generated code can not perfectly reproduce the input UI design because of some particular layout attributes of UI elements, such as float and centring. Third, the coordinates and sizes of elements should be refined in the way that transfers some fixed numbers into relative numbers, such as replacing *width:200px* with *width:50%*, to achieve a more adjustable design that is in line with real UI development.

6.4 Summary

This chapter introduces the code generation part of UI2CODE, which functions as the final output layer of this work to produce the deliverable code. To this end, I proposed a pipeline that first hierarchically segments the entire UI into layout blocks on the basis of cutting lines and gaps between elements, and then assembles all the detection and segmentation results from above steps to a DOM tree to generate the final web program. Nevertheless, the code generation part is not mature enough and requires further researches to address the aforementioned issues.

Chapter 7

Results

This chapter presents some results of the UI2CODE. As mentioned in the previous text, the code generation part is yet unrobust, so I focus on demonstrating the detection results of the UI components detector and evaluating its performance with respect to the accuracy of localization and classification.

7.1 Evaluation

To evaluate the effectiveness of the UI component detector, I test them on the collected datasets of different types.

The classifier is implemented by a simple four-layer CNN, which is trained on a fairly large size of UI component datasets. It is trained on various GUI components shown in table 7.1, in which I used 80% as training data and 20% as testing data. The confusion matrix is presented in Figure 7.1.

Class Name	Number
Input Box	1632
Text	16406
Button	6604
Icon	3601
Img	29991

Table 7.1: Training data of classifier

To judge the effectiveness of the localization part of the UI components detection, I calculate the Intersection-over-union (IoU) between the ground truth and the prediction bounding box. Due to the high demand for precision property of GUI design, as mentioned in section 3.1, the IoU value should be more than 0.8 if two boxes are considered matching.

Figure 7.2 presents the evaluation of the localization by the graphical components detector with respect to the accuracy and recall. We can observe several phenomena from it: (1) The detector's performance on web UI is proven best. The reason for that is the web screenshot is less compact, and there are more gaps among elements so that they are easier to segment, while the mobile UI's layout is tighter hence harder

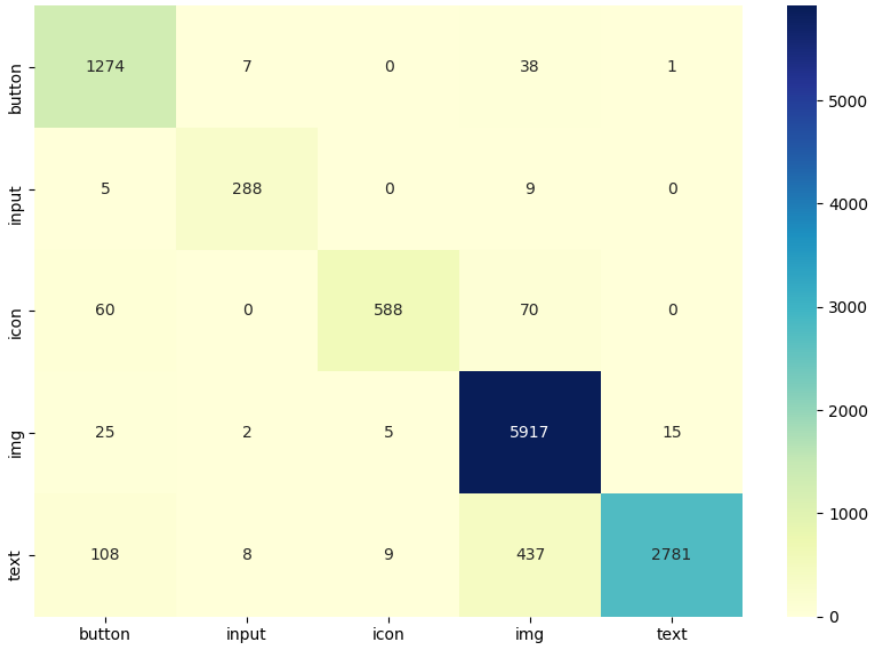


Figure 7.1: The confusion matrix of the CNN classifier from which we calculate the *recall:0.937*, *accuracy:0.920*, and the *balanced accuracy:0.912*

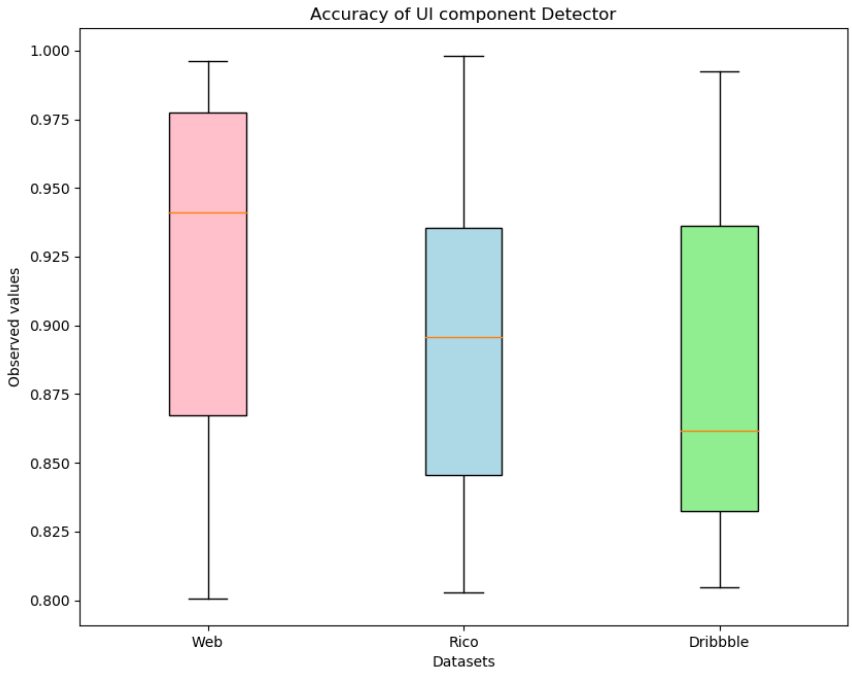
to perform accurate connected component labelling. (2) The overall recall is better than accuracy. It attributes to the pixel-level image processing approaches that cover all pixels in the image whereby few objects would be omitted.

Regarding the processing time, it varies with the sizes of input image as well as the complexity of UI. I conducted the experiment on a windows machine with Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz (8 CPUs) and NVIDIA GeForce GTX 1050 Ti. Typically, an 800 x 600 artistic design drawing takes average 26s to perform UI components detection and text recognition; a 2000 x 1300 real web screenshot takes average 57s to process; a 1000 x 800 mobile UI takes average 32s to process.

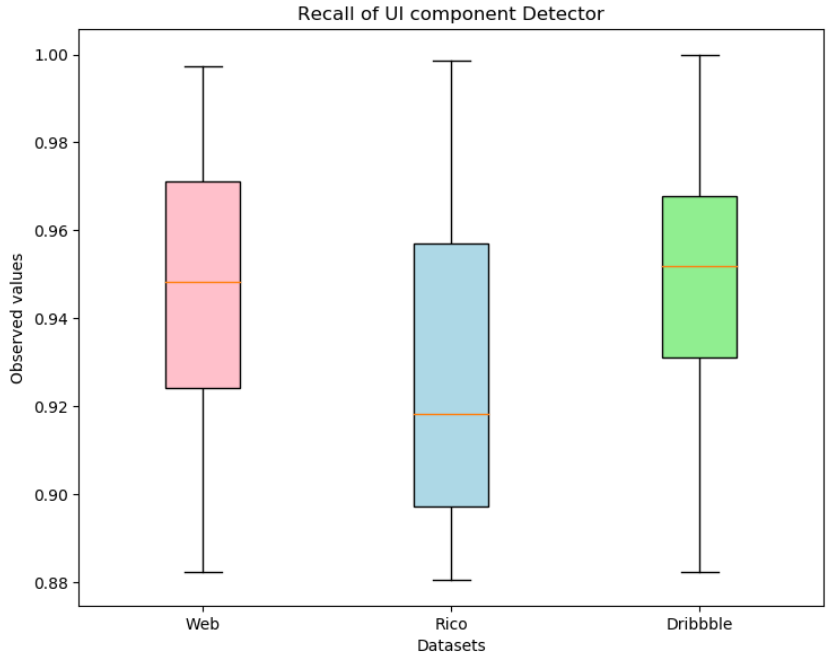
7.2 Results Demonstration

This section demonstrates the detection results on randomly selected UIs from the three datasets mentioned in chapter 4. The thick colourful bounding boxes with labels are predictions given by graphical component detector while the slender green bounding boxes are text recognition results of the CTPN.

Theoretically, the size of the input image can be arbitrary, which only affects the processing time. For the sake of display, I resized and clipped some pictures. The average processing time of these presented images is stated in the captions.



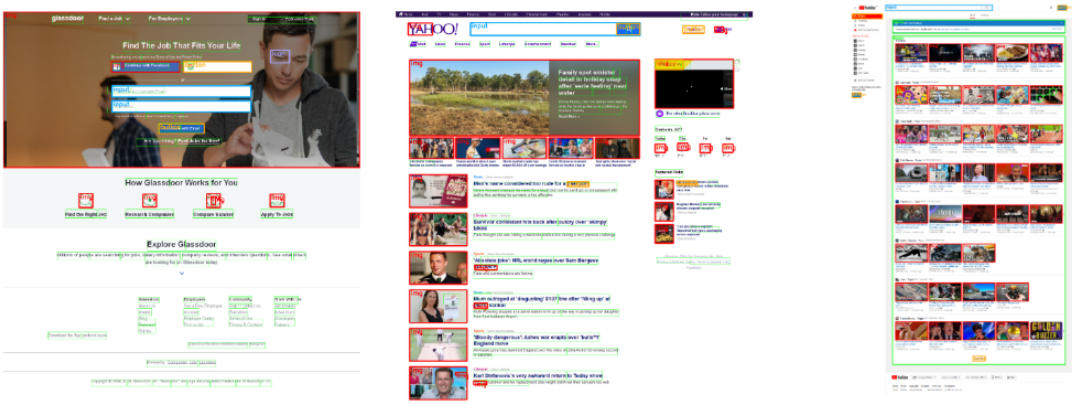
(a) Accuracy



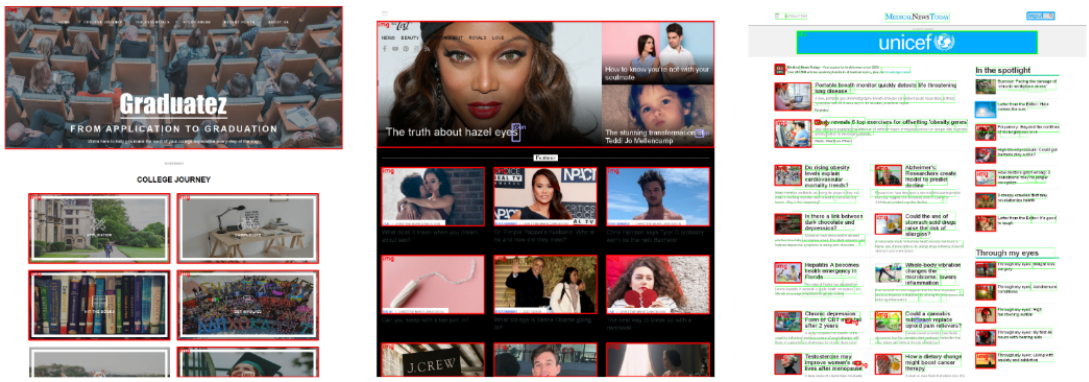
(b) Recall

Figure 7.2: Evaluation of UI graphical component detector.

(a)



(b)

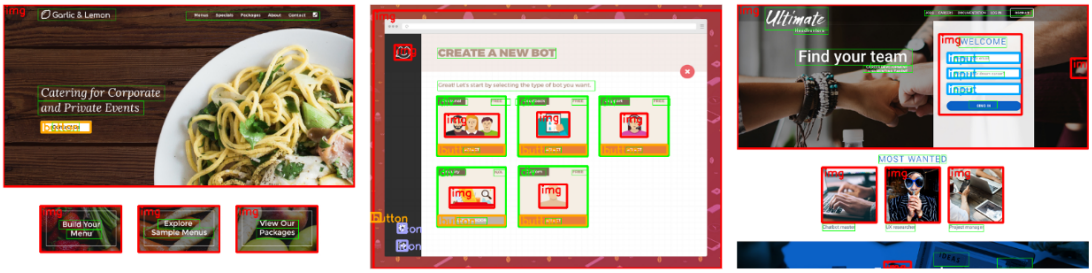


(c)

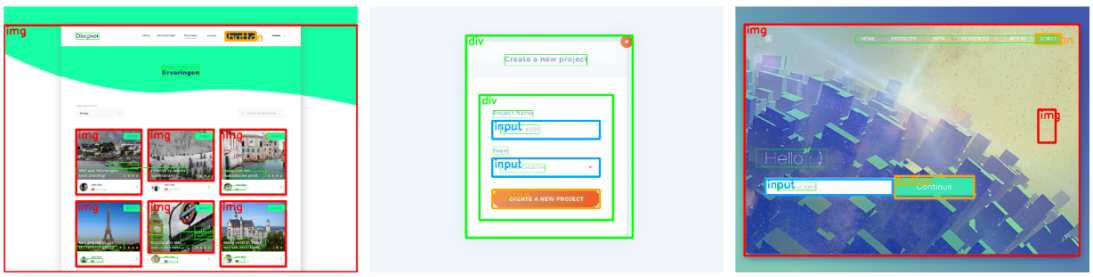
Figure 7.3: UI component detection on real web UI screenshots. Average processing time: 57s.

(a)

(b)



(c)

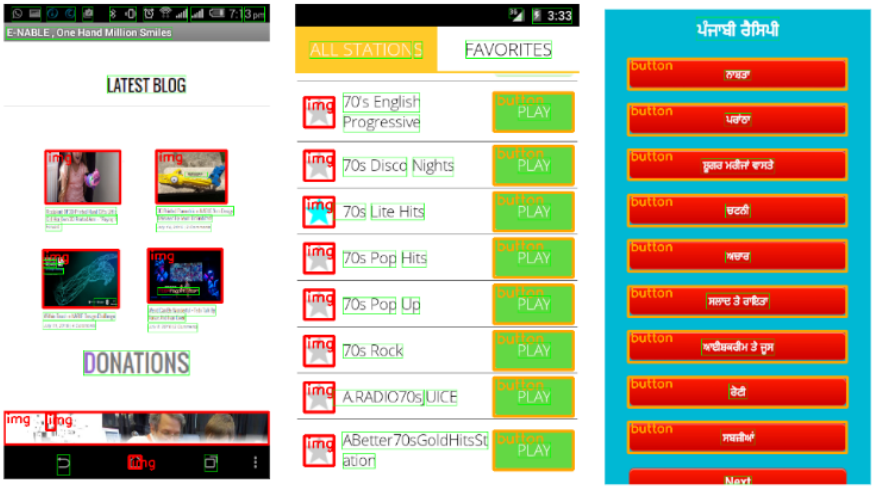


(d)

Figure 7.4: UI component detection on artistic UI design drawing from Dribbble. Average processing time: 28s.

(a)

(b)



(c)

Figure 7.5: UI component detection on real mobile UI screenshots from GooglePlay and Rico. Average processing time: 36s.

Chapter 8

Conclusion

8.1 Conclusion

This thesis introduces a novel computer vision based user interface reverse engineering technique, UI2CODE. The major objective of this system is to bridge the gap between the conceptual design and the code implementation in graphical human-computer interface development, hence boosting the efficiency of the workflow and relieving pains of designers and developers. In detail, UI2CODE takes the UI image, either real screenshot or design drawing, as input and automatically identifies UI components whereby manufacturing the usable and maintainable source code. Unlike existing UI development tools, such as Wix and Wordpress, which focus attention on supporting the fancy design and convenient drag and drop editor but do not offer high-quality source code, UI2CODE provides users with easy-to-use code with interfaces for further development. Thus, it is more suitable for professional projects where full knowledge and ownership are required.

To this end, I built this system as a pipeline consisting of two high-level modules: UI components detector and code generator. Besides, I applied three datasets, including a repository of real web UI screenshot, a published large mobile application database Rico and a self-built dataset of artistic UI design drawings. These data are used to observe the attributes of graphical user interfaces and evaluate UI2CODE. The graphical user interface has various characteristics distinctly different from nature pictures, including heterogeneous contents, picked components, arbitrary-shaped elements and strict demand for precision. These properties hamper deep learning based object detection approaches from performing effective prediction.

Therefore I leveraged the conventional computer vision and image processing methods to implement the UI component detection part to fulfil the particular requirements. Two separate branches are involved in achieving the graphical component detection and text recognition. The graphical component detector comprises multiple novel algorithms and embeds them into three steps: (1) pre-processing that produces the distinct binary map of foreground and background; (2) component detection part extracting and segmenting the connected components as candidates; (3) a classifier based on convolutional neural network to categorize the selected components. On the other hand, I adopted CTPN to perform text recognition on the UI. In

the end, the results are integrated to deliver the final result.

Then a code generator takes the detection result and produces the deliverable front-end program that implements the identical visual effect and expected functionalities of the input UI image. This part is composed of two sections: (1) hierarchical layout block segmentation dividing the image into multiple clusters on the basis of cutting lines; (2) web code generator that first converts the blocks into HTML DOM tree and then transforms this tree into HTML/CSS code in which unique IDs are assigned to elements and interfaces are created for future extension. Finally, UI2CODE exports the high-quality code to the user in short timeframe.

8.2 Future Work

This work has not been finalized yet, and few problems and challenges still exist in each part.

One significant defect of current datasets is the lack of appropriate annotation. In other words, a big proportion of images have no information about their contents, such as the classes and position of components. This problem is caused by the issues of the crawling agent, which is mentioned in section 4.1.2. So the solution could be improving or exploiting a new crawler or combining human power as supplementary.

The UI components detector can well handle most real UI designs, but it suffers the noise caused by false positive detection of small objects. Those noises always come from the isolated content of the image element and are treated as UI components by mistake. I have some clues to settle this problem inspired by a non-maximum suppression extension work [Rothe et al., 2015]. Besides, the layout alignment assumption in artificial UI should also be utilized to refine the results in some ways.

Furthermore, the performance of the text recognition model CTPN is not as accurate as of the UI components detector so we might consider finding some new technique to enhance the precision of text detection.

The code generator can only produce HTML/CSS code so far, but the functionalities are always supported by Javascript program. Besides, there is a huge potential for improvement of the generated code to better accord with the professional product. We can achieve by co-operating with some specialists to gain their feedback and comments.

Finally, the quantitative comparison between UI components detector of UI2CODE and deep learning object detection methods should be drawn after we improve the datasets by adding more annotations of images.

Bibliography

2019. Block-level elements. https://developer.mozilla.org/en-US/docs/Web/HTML/Block-level_elements. (cited on page 63)

2019. Tree (data structure). https://en.wikipedia.org/wiki/Tree_(data_structure). (cited on page 65)

Alharbi, K. and Yeh, T., 2015. Collect, decompile, extract, stats, and diff: Mining design pattern changes in android apps. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '15 (Copenhagen, Denmark, 2015), 515–524. ACM, New York, NY, USA. doi:10.1145/2785830.2785892. http://doi.acm.org/10.1145/2785830.2785892. (cited on pages 10 and 26)

Asa Ben-Hur, H. T. S. V. V., David Horn, 2001. Support vector clustering. *Journal of Machine Learning Research*, (2001), 125–137. http://www.jmlr.org/papers/v2/horn01a.html. (cited on page 49)

Basten, B., 2011. *Ambiguity detection for programming language grammars*. s.n. (cited on page 23)

Brown, C. M., 1999. *Human-computer interface design guidelines*. Intellect. (cited on pages 1 and 9)

Campos, P. and Nunes, N., 2007. Practitioner tools and workstyles for user-interface design. *IEEE Software*, 24, 1 (2007), 73–80. doi:10.1109/ms.2007.24. (cited on page 9)

Canny, J., 1986. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8, 6 (Nov 1986), 679–698. doi:10.1109/TPAMI.1986.4767851. (cited on pages 9, 11, and 31)

Canny, J., 1987. A computational approach to edge detection. *Readings in Computer Vision*, (1987), 184–203. doi:10.1016/b978-0-08-051581-6.50024-6. (cited on page 17)

Cepelewicz, J., 2019. Where we see shapes, ai sees textures. https://www.quantamagazine.org/where-we-see-shapes-ai-sees-textures-20190701. (cited on page 29)

Cortes, C. and Vapnik, V., 1995. Support-vector networks. *Machine Learning*, 20, 3 (Sep 1995), 273–297. doi:10.1007/BF00994018. https://doi.org/10.1007/BF00994018. (cited on page 47)

Cosulschi, M.; Giurca, A.; Udrescu, B.; Constantinescu, N.; and Gabroveanu, M., 2006. Html pattern generator–automatic data extraction from web pages. In *2006 Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 75–78. doi:10.1109/SYNASC.2006.43. (cited on pages 4, 11, and 65)

Coyette, A.; Kieffer, S.; and Vanderdonckt, J., 2007. Multi-fidelity prototyping of user interfaces. *Lecture Notes in Computer Science Human-Computer Interaction – INTERACT 2007*, (2007), 150–164. doi:10.1007/978-3-540-74796-3_16. (cited on page 9)

Dalal, N. and Triggs, B., 2005. Histograms of Oriented Gradients for Human Detection. In *International Conference on Computer Vision & Pattern Recognition (CVPR '05)*, vol. 1, 886–893. IEEE Computer Society, San Diego, United States. doi:10.1109/CVPR.2005.177. https://hal.inria.fr/inria-00548512. (cited on pages 48 and 49)

Deka, B.; Huang, Z.; Franzen, C.; Hibschman, J.; Afergan, D.; Li, Y.; Nichols, J.; and Kumar, R., 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17 (Québec City, QC, Canada, 2017), 845–854. ACM, New York, NY, USA. doi:10.1145/3126594.3126651. http://doi.acm.org/10.1145/3126594.3126651. (cited on pages 9, 10, and 26)

Deka, B.; Huang, Z.; and Kumar, R., 2016. Erica: Interaction mining mobile apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16 (Tokyo, Japan, 2016), 767–776. ACM, New York, NY, USA. doi:10.1145/2984511.2984581. http://doi.acm.org/10.1145/2984511.2984581. (cited on page 26)

Dev, O., 2014. Structural analysis and shape descriptors. https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#id5. (cited on pages 11 and 40)

Dillencourt, M. B.; Samet, H.; and Tamminen, M., 1992. A general approach to connected-component labeling for arbitrary image representations. *J. ACM*, 39, 2 (Apr. 1992), 253–280. doi:10.1145/128749.128750. http://doi.acm.org/10.1145/128749.128750. (cited on page 36)

Dixon, M. and Fogarty, J., 2010. Prefab. *Proceedings of the 28th international conference on Human factors in computing systems - CHI 10*, (2010). doi:10.1145/1753326.1753554. (cited on page 9)

Douglas, D. and Peucker, T., 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, (1973), 112–122. doi:https://utpjournals.press/doi/10.3138/FM57-6770-U75U-7727. (cited on pages 11 and 40)

Dribbble, 2019. Discover the world's top designers creatives. https://dribbble.com/. (cited on page 10)

Duda, R. O. and Hart, P. E., 1972. Use of the hough transformation to detect lines and curves in pictures. *Comm. ACM, Vol. 15,*, (01 1972), 11–15. (cited on pages 5, 11, 40, and 59)

D'Haen, J.; den Poel, D. V.; Thorleuchter, D.; and Benoit, D., 2016. Integrating expert knowledge and multilingual web crawling data in a lead qualification system. *Decision Support Systems*, 82 (2016), 69 – 78. doi:https://doi.org/10.1016/j.dss.2015.12.002. http://www.sciencedirect.com/science/article/pii/S016792361500216X. (cited on pages 5, 10, and 11)

EGGLESTON, P., 2015. Understanding oversegmentation and region merging. https://www.vision-systems.com/non-factory/security-surveillance-transportation/article/16739494/understanding-oversegmentation-and-region-merging. (cited on page 18)

Engel, A., 1997. *Problem Solving Strategies*. Springer-Verlag New York, Incorporated. (cited on page 61)

Farrugia, L. J., 2007. *ORTEP*-3 for Windows - a version of *ORTEP*-III with a Graphical User Interface (GUI). *Journal of Applied Crystallography*, 30, 5 Part 1 (Oct 2007), 565. doi:10.1107/S0021889897003117. https://doi.org/10.1107/S0021889897003117. (cited on page 1)

Felzenszwalb, P. F.; Girshick, R. B.; McAllester, D.; and Ramanan, D., 2010. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32, 9 (Sep. 2010), 1627–1645. doi:10.1109/TPAMI.2009.167. (cited on page 16)

Freedman, D., 2012. *Statistical models: theory and practice*. Cambridge University Press. (cited on page 14)

Freeman, W. T. and Roth, M., 1994. Orientation histograms for hand gesture recognition. Technical Report TR94-03, MERL - Mitsubishi Electric Research Laboratories, Cambridge, MA 02139. https://www.merl.com/publications/TR94-03/. (cited on pages 11, 47, and 48)

Fu, B.; Lin, J.; Li, L.; Faloutsos, C.; Hong, J.; and Sadeh, N., 2013. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13 (Chicago, Illinois, USA, 2013), 1276–1284. ACM, New York, NY, USA. doi:10.1145/2487575.2488202. http://doi.acm.org/10.1145/2487575.2488202. (cited on page 10)

Gandhi, R., 2018. R-cnn, fast r-cnn, faster r-cnn, yolo - object detection algorithms. https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e. (cited on pages 13 and 16)

Geirhos, R., 2018. Out of shape? why deep learning works differently. https://blog.usejournal.com/why-deep-learning-works-differently-than-we-thought-ec28823bdbc. (cited on page 29)

Geirhos, R.; Rubisch, P.; Michaelis, C.; Bethge, M.; Wichmann, F. A.; and Brendel, W., 2019. Imagenet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bygh9j09KX. (cited on page 29)

Girshick, R., 2015. Fast r-cnn. In *The IEEE International Conference on Computer Vision (ICCV)*. (cited on pages 10, 13, and 53)

Girshick, R.; Donahue, J.; Darrell, T.; and Malik, J., 2014a. Rich feature hierarchies for accurate object detection and semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (cited on pages 10 and 13)

Girshick, R.; Donahue, J.; Darrell, T.; and Malik, J., 2014b. Rich feature hierarchies for accurate object detection and semantic segmentation. (cited on page 14)

GmbH, 2019. Websitebuilders. https://www.top10best-websitebuilders.com/au/website-builders/?gclid=CjwKCAjwlovtBRBrEiwAG3XJ-4QfLLJb3Rg_BQWCV-5-EiMfFEgpgozldPUNwjcvx4tNITj_4HH-9hoCE9AQAvD_BwE. (cited on page 2)

Gordiyenko, S., 2019a. Website development process: Full guide in 7 steps. https://xbsoftware.com/blog/website-development-process-full-guide/. (cited on page 1)

Gordiyenko, S., 2019b. Website development process: Full guide in 7 steps. https://xbsoftware.com/blog/website-development-process-full-guide/. (cited on page 7)

Graham, R. M.; Knuth, D. E.; and Patashnik, O., 1994. *Concrete mathematics*. Addison-Wesley. (cited on page 61)

Graves, A. and Schmidhuber, J., 2005. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18, 5 (2005), 602 – 610. doi:https://doi.org/10.1016/j.neunet.2005.06.042. http://www.sciencedirect.com/science/article/pii/S0893608005001206. IJCNN 2005. (cited on page 54)

Gu, J.; Wang, Z.; Kuen, J.; Ma, L.; Shahroudy, A.; Shuai, B.; Liu, T.; Wang, X.; Wang, G.; Cai, J.; and Chen, T., 2018. Recent advances in convolutional neural networks. *Pattern Recognition*, 77 (2018), 354 – 377. doi:https://doi.org/10.1016/j.patcog.2017.10.013. http://www.sciencedirect.com/science/article/pii/S0031320317304120. (cited on page 16)

Gupta, A.; Vedaldi, A.; and Zisserman, A., 2016. Synthetic data for text localisation in natural images. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (cited on page 53)

Gupta, S.; Kaiser, G.; Neistadt, D.; and Grimm, P., 2003. Dom-based content extraction of html documents. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03 (Budapest, Hungary, 2003), 207–214. ACM, New York, NY, USA. doi:10.1145/775152.775182. http://doi.acm.org/10.1145/775152.775182. (cited on pages 4, 11, and 65)

Harris, A., 2014. *HTML5 and CSS3 all-in-one for dummies*. John Wiley Sons, Inc. (cited on pages 57 and 61)

He, K.; Gkioxari, G.; Dollár, P.; and Girshick, R. B., 2017. Mask R-CNN. *CoRR*, abs/1703.06870 (2017). http://arxiv.org/abs/1703.06870. (cited on page 13)

Hochreiter, S. and Schmidhuber, J., 1997. Long short-term memory. *Neural Computation*, 9, 8 (1997), 1735–1780. doi:10.1162/neco.1997.9.8.1735. https://doi.org/10.1162/neco.1997.9.8.1735. (cited on page 54)

Huizinga, D. and Kolawa, A., 2007. Automated defect prevention best practices in software management. https://www.amazon.com/Automated-Defect-Prevention-Practices-Management/dp/0470042125. (cited on page 22)

Jacobs, D., 2005. Image gradients. *Class Notes for CMSC*, (9 2005). http://www.cs.umd.edu/~djacobs/CMSC426/ImageGradients.pdf. (cited on page 32)

Jeeva, M., 2018. The scuffle between two algorithms -neural network vs. support vector machine. https://medium.com/analytics-vidhya/the-scuffle-between-two-algorithms-neural-network-vs-support-vector-machine-16abe0eb4181. (cited on page 51)

Kobayashi, M. and Takeda, K., 2000. Information retrieval on the web. *ACM Comput. Surv.*, 32, 2 (Jun. 2000), 144–173. doi:10.1145/358923.358934. http://doi.acm.org/10.1145/358923.358934. (cited on pages 21 and 23)

Landay, J. A. and Myers, B. A., 1994. Interactive sketching for the early stages of user interface design. (1994). doi:10.21236/ada285339. (cited on page 9)

Landay, J. A. and Myers, B. A., 2001. Sketching interfaces: toward more human interface design. *Computer*, 34, 3 (March 2001), 56–64. doi:10.1109/2.910894. (cited on page 9)

Laurel, B. and Mountford, S. J. (Eds.), 1990. *The Art of Human-Computer Interface Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0201517973. (cited on pages 1 and 9)

Le, D.; Thoma, G. R.; and Zou, J., 2006. Combining dom tree and geometric layout analysis for online medical journal article segmentation. In *Proceedings of the 6th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL '06)*, 119–128. doi:10.1145/1141753.1141777. (cited on page 65)

Lee, S.; Kwak, S.; and Cho, M., 2019. Universal bounding box regression and its applications. *CoRR*, abs/1904.06805 (2019). http://arxiv.org/abs/1904.06805. (cited on pages 10 and 14)

Lee, X., 2012. Jargons explained: Tag, element, node, object, attribute, property, method. http://xahlee.info/js/javascript_DOM_confusing_terminology.html. (cited on pages 5, 61, 63, and 66)

Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.-Y.; and Berg, A. C., 2016. Ssd: Single shot multibox detector. In *Computer Vision – ECCV 2016*, 21–37. Springer International Publishing, Cham. (cited on pages 4, 10, and 13)

Lowe, D. G., 1999. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, 1150–1157 vol.2. doi:10.1109/ICCV.1999.790410. (cited on pages 11, 47, and 50)

Lowe, D. G., 2004. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60, 2 (Nov 2004), 91–110. doi:10.1023/B:VISI.0000029664.99615.94. https://doi.org/10.1023/B:VISI.0000029664.99615.94. (cited on pages 11 and 47)

McConnell, R. K., 1986. Method of and apparatus for pattern recognition. https://patents.google.com/patent/US4567610. (cited on pages 11 and 47)

MDN, 2019. Block-level elements. https://developer.mozilla.org/en-US/docs/Web/HTML/Block-level_elements. (cited on page 58)

Mehmet Sezgin, B. S., 2004. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging*, , 13 (2004), 146 – 165. doi:https://doi.org/10.1117/1.1631315. (cited on page 33)

Mikolajczyk, K. and Schmid, C., 2005. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27, 10 (Oct 2005), 1615–1630. doi:10.1109/TPAMI.2005.188. (cited on page 50)

Moizuddin, K., 2019. Components of the selenium automation tool - dzone devops. https://dzone.com/articles/components-of-selenium-automation-tool. (cited on pages 22 and 23)

Mordvintsev, A. and Revision, A. K., 2013. Introduction to sift (scale-invariant feature transform)¶. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html. (cited on page 50)

Musciano, C. and Kennedy, B., 2007. *HTML XHTML: the definitive guide*. OReilly. (cited on pages 11, 61, 65, 66, and 68)

Nguyen, T. A. and Csallner, C., 2015. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 248–259. doi:10.1109/ASE.2015.32. (cited on pages 7 and 9)

Olston, C. and Najork, M., 2010. Web crawling. *Foundations and Trends® in Information Retrieval*, 4, 3 (2010), 175–246. doi:10.1561/1500000017. http://dx.doi.org/10.1561/1500000017. (cited on pages 5, 10, and 11)

Patel, S., 2017. Chapter 2 : Svm (support vector machine) - theory. https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72. (cited on pages 11 and 49)

Prewit, J., 1970. Object enhancement and extraction. In *Picture Processing and Psychopictorics*, 75–120. B.S. Lipkin. (cited on page 32)

R., M. S. and P., B. B., 2014. Searching for Inspiration: An In-Depth Look at Designers Example Finding Practices. vol. Volume 7: 2nd Biennial International Conference on Dynamics for Design; 26th International Conference on Design Theory and Methodology of *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. doi:10.1115/DETC2014-35450. https://doi.org/10.1115/DETC2014-35450. V007T07A035. (cited on page 9)

Redmon, J.; Divvala, S.; Girshick, R.; and Farhadi, A., 2016. You only look once: Unified, real-time object detection. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (cited on pages 4, 10, 13, and 29)

Redmon, J. and Farhadi, A., 2018. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767 (2018). http://arxiv.org/abs/1804.02767. (cited on pages 10 and 29)

Ren, S.; He, K.; Girshick, R.; and Sun, J., 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems 28* (Eds. C. Cortes; N. D. Lawrence; D. D. Lee; M. Sugiyama; and R. Garnett), 91–99. Curran Associates, Inc. http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf. (cited on pages 4, 10, 13, 17, 29, and 53)

Richardson, L., 2015. Beautiful soup documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/. (cited on page 22)

Riva, M. d. l., 2019. 32 user interface elements for ui designers: Your ultimate glossary. https://careerfoundry.com/en/blog/ui-design/ui-element-glossary/. (cited on pages 23 and 24)

Roberts, L., 1963. Machine perception of three-dimensional solids. (1963). https://dspace.mit.edu/bitstream/handle/1721.1/11589/33959125-MIT.pdf. (cited on page 32)

Rothe, R.; Guillaumin, M.; and Van Gool, L., 2015. Non-maximum suppression for object detection by passing messages between windows. In *Computer Vision – ACCV 2014*, 290–306. Springer International Publishing, Cham. (cited on page 76)

SACHAN, A., 2017. Zero to hero: Guide to object detection using deep learning: Faster r-cnn,yolo,ssd. https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/. (cited on page 17)

Sahami Shirazi, A.; Henze, N.; Schmidt, A.; Goldberg, R.; Schmidt, B.; and Schmauder, H., 2013. Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13 (London, United Kingdom, 2013), 275–284. ACM, New York, NY, USA. doi:10.1145/2494603.2480308. http://doi.acm.org/10.1145/2494603.2480308. (cited on pages 10 and 26)

Samet, H. and Tamminen, M., 1988. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10, 4 (July 1988), 579–586. doi:10.1109/34.3918. (cited on page 36)

SEAL, H. L., 1967. Studies in the History of Probability and Statistics. XV The historical development of the Gauss linear model. *Biometrika*, 54, 1-2 (06 1967), 1–24. doi:10.1093/biomet/54.1-2.1. https://doi.org/10.1093/biomet/54.1-2.1. (cited on page 14)

Selene M., B., 2018. 6 phases of the web site design and development process. https://www.idesignstudios.com/web-design/phases-web-design-development-process/. (cited on page 1)

Shapiro, S. G. C., Linda G., 2002. In *Computer Vision*. B.S. Lipkin. (cited on page 33)

Sobel, I., 1968. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, (02 1968). (cited on page 32)

Suzuki, S. and be, K., 1985. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30, 1 (1985), 32 – 46. doi:https://doi.org/10.1016/0734-189X(85)90016-7. http://www.sciencedirect.com/science/article/pii/0734189X85900167. (cited on page 32)

Team, O. D., 2012. Structural analysis and shape descriptors. https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#structural-analysis-and-shape-descriptors. (cited on pages 32 and 38)

Tian, Z.; Huang, W.; He, T.; He, P.; and Qiao, Y., 2016. Detecting text in natural image with connectionist text proposal network. In *Computer Vision – ECCV 2016*, 56–72. Springer International Publishing, Cham. (cited on pages 4, 29, and 52)

Tutorialspoint, 2016. Html - blocks. https://www.tutorialspoint.com/html/html_blocks.htm. (cited on pages 58 and 62)

Uijlings, J. R. R.; van de Sande, K. E. A.; Gevers, T.; and Smeulders, A. W. M., 2013. Selective search for object recognition. *International Journal of Computer Vision*, 104, 2 (Sep 2013), 154–171. doi:10.1007/s11263-013-0620-5. https://doi.org/10.1007/s11263-013-0620-5. (cited on page 16)

Vincent, L. and Soille, P., 1991. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13, 6 (June 1991), 583–598. doi:10.1109/34.87344. (cited on page 36)

W3Cschool. https://www.w3schools.com/css/css_border.asp. (cited on page 59)

W3Cschool, 2019. https://www.w3schools.com/css/css_boxmodel.asp. (cited on page 11)

WHATWG, 2019. Html. https://html.spec.whatwg.org/. (cited on pages 57 and 61)

White, T. D.; Fraser, G.; and Brown, G. J., 2019. Improving random gui testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019 (Beijing, China, 2019), 307–317. ACM, New York, NY, USA. doi:10.1145/3293882.3330551. http://doi.acm.org/10.1145/3293882.3330551. (cited on pages 3, 10, and 57)

Whitmer, R., 2009. Document object model (dom). https://www.w3.org/DOM/#what. (cited on pages 5, 10, 11, 21, and 65)

Yu, S., 2019. Svm - theory. https://zhuanlan.zhihu.com/p/31886934. (cited on page 49)

Zeidler, C.; Lutteroth, C.; Stuerzlinger, W.; and Weber, G., 2013. The auckland layout editor: An improved gui layout specification process. 343–352. doi:10.1145/2501988.2502007. (cited on page 7)

Zhang, Y.; Chen, Y.; Huang, C.; and Gao, M., 2019. Object detection network based on feature fusion and attention mechanism. *Future Internet*, 11, 1 (Feb 2019), 9. doi:10.3390/fi11010009. (cited on page 17)

Zhang, Z.; Zhang, C.; Shen, W.; Yao, C.; Liu, W.; and Bai, X., 2016. Multi-oriented text detection with fully convolutional networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (cited on page 53)