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
7 October 2019

Acknowledgments

Who do you want to thank?

Abstract

UI designs consist of heterogeneous UI elements, such as natural images, drawings, graphic symbols, and computer-rendered objects (e.g., texts, buttons, boxes). Existing image segmentation techniques start with a fine partition into small regions, and gradually merge them into larger and larger ones. When applied to UI designs, this bottom-up strategy results in many primitive low-level shapes (i.e., oversegmentation). We propose a novel, top-down, divide-and-conquer approach that segments a UI design into a tree of UI design elements, corresponding intuitively to the perceptual boundary and organization of UI design elements. Experiments on large numbers of UI designs from Google play and Dribbble confirm the effectiveness of our approach for segmenting UI designs with complex visual composition of heterogeneous UI design elements. Our approach enables a new way of data-driven design applications on Google play and Dribbble design data, which is impossible before due to the lack of element-level metadata.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Statement

I believe A is better than B.

1.2 Introduction

Put your introduction here. You could use \fix{ABCDEFG.} to leave your comments, see the box at the left side.

You have to rewrite your thesis!!!

1.3 Thesis Outline

How many chapters you have? You may have Chapter 2, Chapter **??**, Chapter **??**, Chapter 6, and Chapter 7.

Chapter 2

Background and Related Work

At the begging of each chapter, please introduce the motivation and high-level picture of the chapter. You also have to introduce sections in the chapter.

Section 2.1 xxxx.

Section 2.2 yyyy.

2.1 Motivation

2.2 Related work

You may reference other papers. For example: Generational garbage collection [Lieberman and Hewitt, 1983; Moon, 1984; Ungar, 1984] is perhaps the single most important advance in garbage collection since the first collectors were developed in the early 1960s. (doi: "doi" should just be the doi part, not the full URL, and it will be made to link to dx.doi.org and resolve. shortname: gives an optional short name for a conference like PLDI '08.)

2.3 Summary

Summary what you discussed in this chapter, and mention the story in next chapter. Readers should roughly understand what your thesis takes about by only reading words at the beginning and the end (Summary) of each chapter.

Chapter 3

Data Collection

First of all, the initial step of this project is to inspect data and to build the datasets for further analyses and experiments. Particularly, to meed the specific goal of this project, the datasets should consist of a variety of images of graphic user interface (GUI) designs, which should include 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.

3.1 Web Page Dataset

3.2 Mobile Application Dataset

Chapter 4

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 implements 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 in graphic interface.

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). In the end, the results from those modules are merged and refined to produce the final result.

Performance of the image processing technique compared to machine learning, 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, which arouses some deep-going thoughts about the nature of deep learning methods, and I elaborate those thinkings in Chapter 5.

4.1 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.

Three categories including totally five types of graphic user interface components are defined in this process: interactive elements *(button, input box)*, static resource *(image, icon)* and layout structure *(block)*, see Table 4.2.

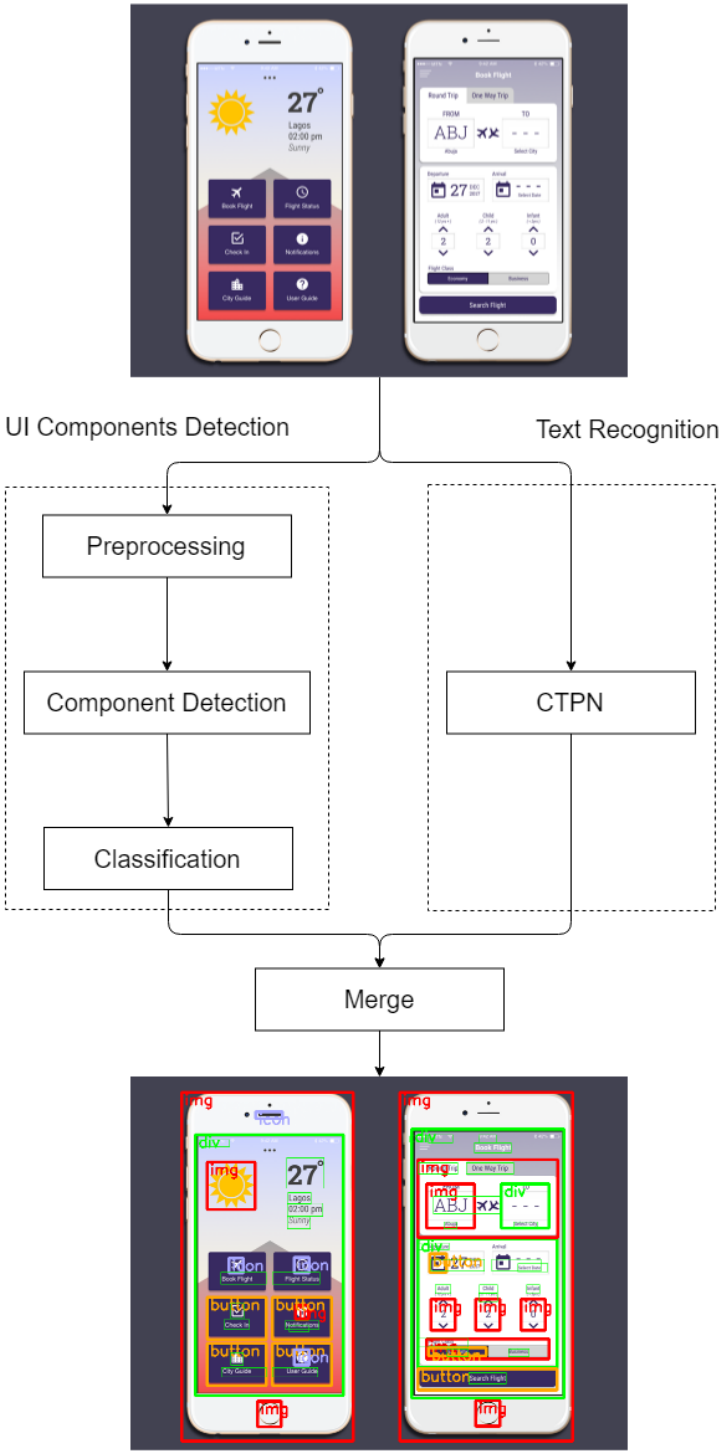


Figure 4.1: The architecture of the UI components detection pipeline. The input image here is a conceptual design drawing of a mobile application, and it is processed by two independent branches to segment the UI components and detect the text regions respectively. Then the results are merged and refined to get the final result.

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 4.1.

4.2 Pre-processing

The first step for of the UI components detection pipeline is pre-processing. It transforms the input image into specific form that is convenient for further processing. Three substeps are conducted here: gray-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 4.2.

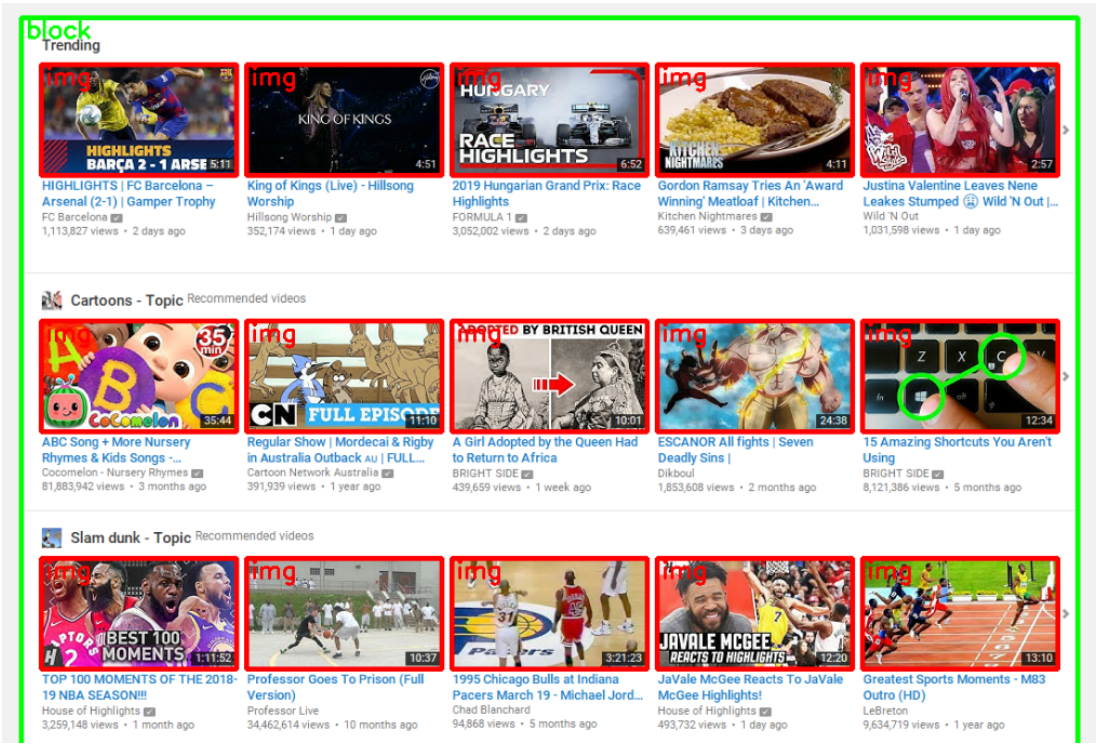


Figure 4.2: A section of screenshot of YouTube website. Plenty of image components (labeled in *img* with red bounding boxes) with colourful contents appear on this user interface, but we want to leave out the information of the real contents and treat them as parts of individual UI components.

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 and findContour method in OpenCV based on techniques proposed by Satoshi et al., 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 4.3. So, I propose a new method to satisfy this purpose.

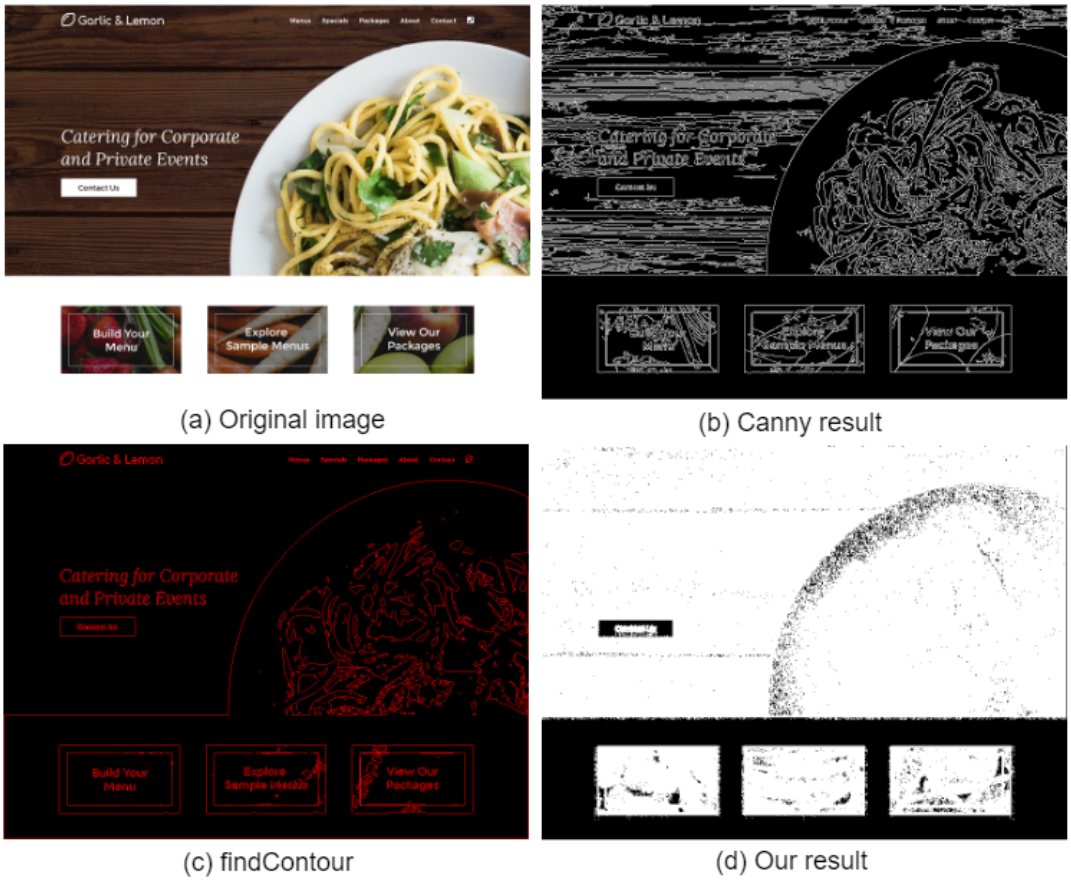


Figure 4.3: The picture (a) is the original image; the image (b) is the result of Canny algorithm, which extracts too many details of texture; the picture (c) is the result of findContour function in OpenCV library, and it focuses on calculation of the boundary of objects; (d) is the binary image processed by our method, which convert the components to a simple binary image consisting of few integrated objects without too many redundant texture information.

4.2.1 Gradient Calculation

The gradient of a digital image measures how each pixel changes in terms of significance and direction. Popular techniques of image gradient calculation are Roberts

cross operator, Prewitt operator, and Sobel operator. 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 gradient by formulae below:

$$gx = \frac{\partial f(x,y)}{\partial x} = f(x+1,y) - f(x,y) \tag{4.1}$$

$$gy = \frac{\partial f(x,y)}{\partial y} = f(x,y+1) - f(x,y) \tag{4.2}$$

$$G(x,y) = |gx| + |gy| \tag{4.3}$$

where: f(x,y) is the pixel vale 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 4.4(b).

4.2.2 Binarization

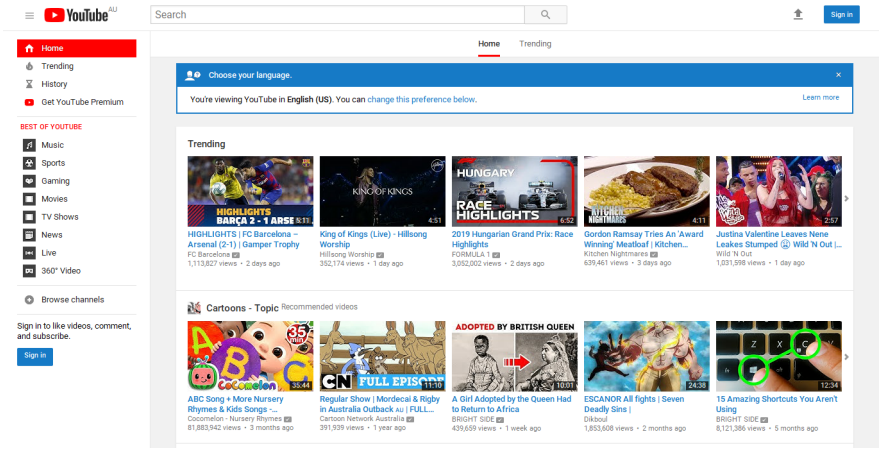
The second substep of pre-processing is to binarize the gray-scale map, the purpose of this process is to intensify the component regions from the background.

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

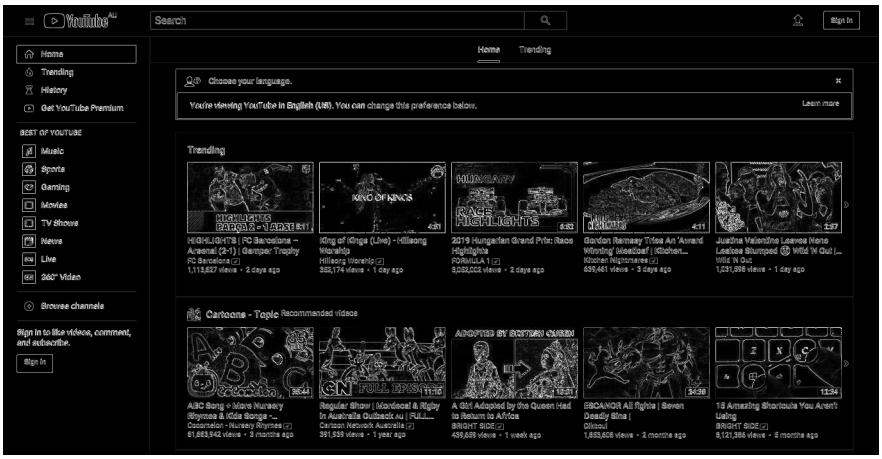
The goal of binarization is to assign a binary value to every pixel in an image. In our case, this step assigns either 255 (white) or 0 (black) to each pixel; points whose value is 255 means could be parts of an interface component; value 0, on the contrary, means this point is part of the background, as demonstrated in Figure 4.4(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.

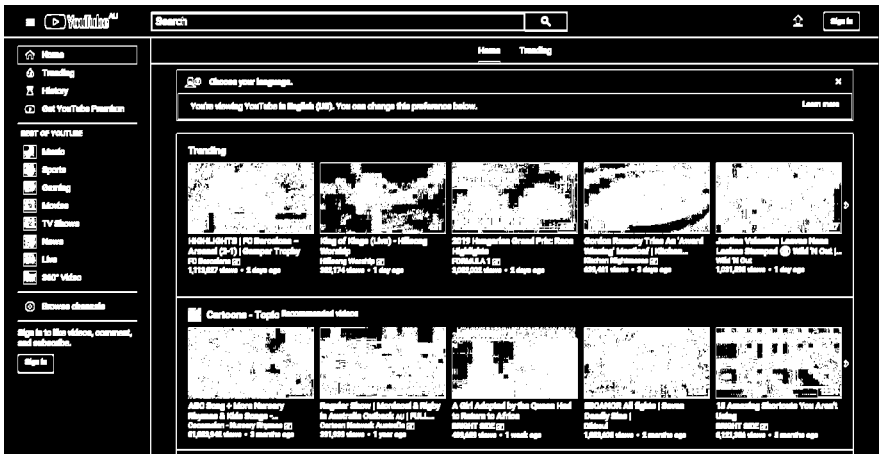
Finally, a binary map that contains clear foreground objects is produced by the pre-processing for further operations.



(a) Original input image



(b) Gradient map



(c) Binary map

Figure 4.4: The visualized demonstration of the pre-processing. The original image Figure 4.4(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 4.4(b); then according to the observation of foreground and background in the human-computer interface, a binary map Figure 4.4(c) is generated

4.3 Component Detection

Based on the acquired binary map, this stage tries to extract and heuristically classify 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 4.4, 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.

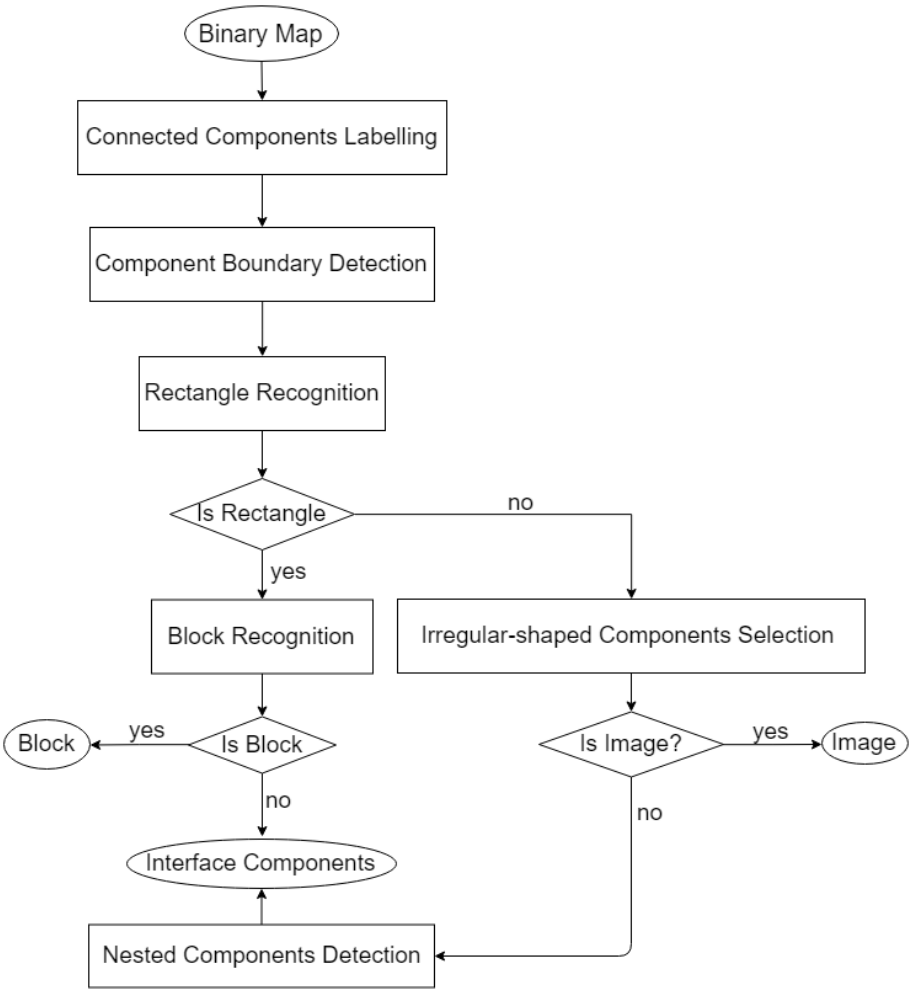


Figure 4.5: Flow chart of the Component Detection pipeline. This pipeline takes binary map as input, and the result of this step consists of three categories of UI elements: *Block, Image and Interface Components*

4.3.1 Connected Components Labelling

This process refers to the connected-component labeling algorithm, which is demonstrated in Figure 4.6, the purpose is to assign each pixel a label identifying the connected component to which this pixel belongs. In other words, this substep aims to segment connected components form the binary image.

I refer to the Seed Filling algorithm in computer graphic to implement my own method. The pseudocode of this technique is shown in Algorithm 4.3.1.

Algorithm 1 Connected-component labeling

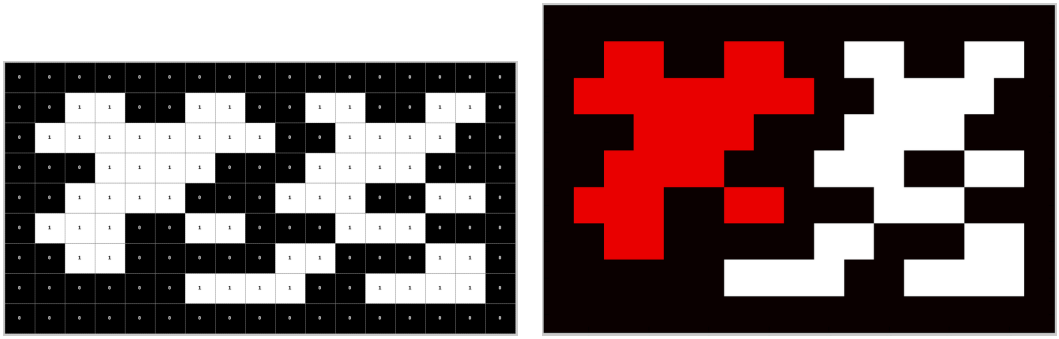
Input: Binary map
Output: An array of components, each component contains a group of points that constitute it

```
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 4.6: Connected-component labelling demonstration

(3) Pop out an element from the store queue, inspect all of its neighbors. If the neighbor is foreground and hasn't been labelled, then add this point into the store queue and count it as a point of 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 above algorithm, the output of this function is an array of components. The way I store a component is define a list of points, in which the points are all foreground and connected.

4.3.2 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 4.3.2, 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 in the loop, then this point should be considered as the new top value at this column in the component, done by $BorderTop[Column] \leftarrow Row$.

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

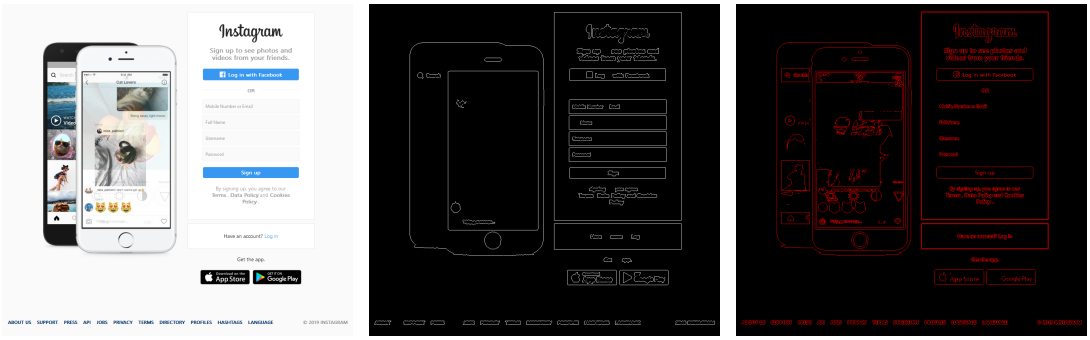
$BorderTop, BorderBottm, BorderLeft, BorderRight \leftarrow \{\ \}, \{\ \}, \{\ \}, \{\ \}$

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 $Boundary$

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, 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 4.7: Demonstration of the Four-border boundary detection. The 4.7(a) is the original input image from a web interface design; the result of this algorithm is shown in figure 4.7(b), which does not contain the fine grained details of the components but only the outer boundaries; the 4.7(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.

4.3.3 Rectangle Recognition

Another observation on human-computer interface is that most of the elements have regular shape. 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.

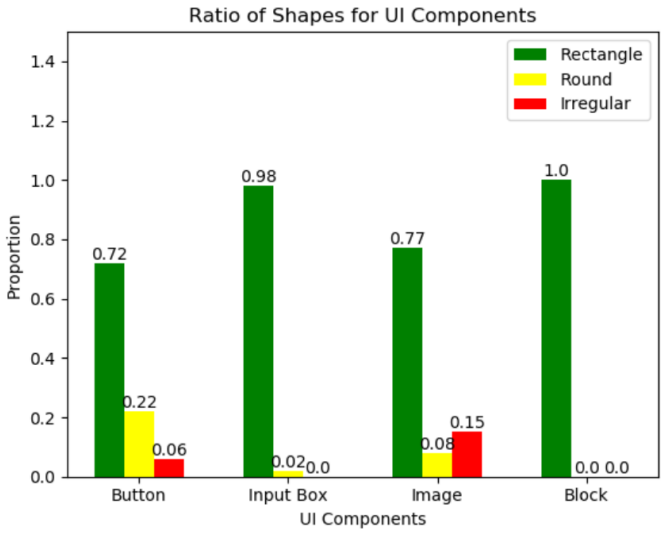


Figure 4.8: The proportion of UI components with different shapes.

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

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

Existing techniques, such as approxPolyDP in OpenCV library and Hough transform, 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 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 boundary. In addition, this method measures the dentation to filter out concave objects. The pseudocode is show in Algorithm 4.3.3.

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.

4.3.4 Block Recognition

We define a bordered region enclosing various multiple elements as block, Figure 4.9 presents two examples. As explained in the section 4.4, 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 easily to be misidentified as image element, especially when the block containing images as shown in Figure 4.9(c).

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

One detailed observation of block is that it can be likened to a wireframe, as demonstrated in Figure 4.9(b) and Figure 4.9(d). In other words, there should a gap between the border lines and the contents in it. Therefore, the algorithm identifying block is designed by detecting the gaps, and it only check the rectangular objects because of the shape attribute. The Python style pseudocode is shown in Algorithm 4.3.4. It just illustrates the basic idea of this algorithm, the real implementation would be more sophisticated because of the complicity of input images.

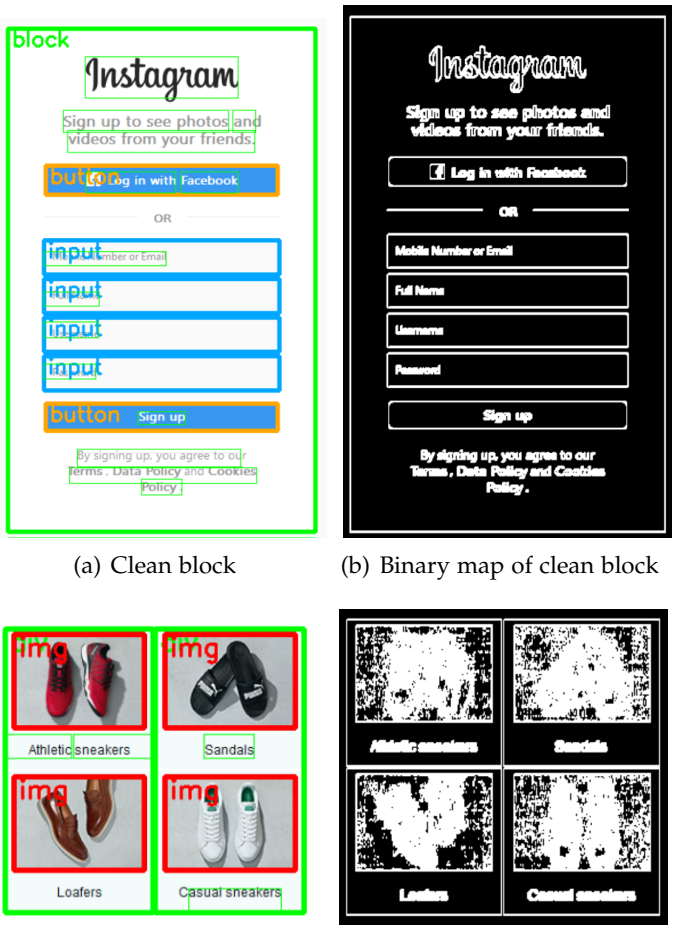
Algorithm 4 Block Recognition

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 as gap.



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

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

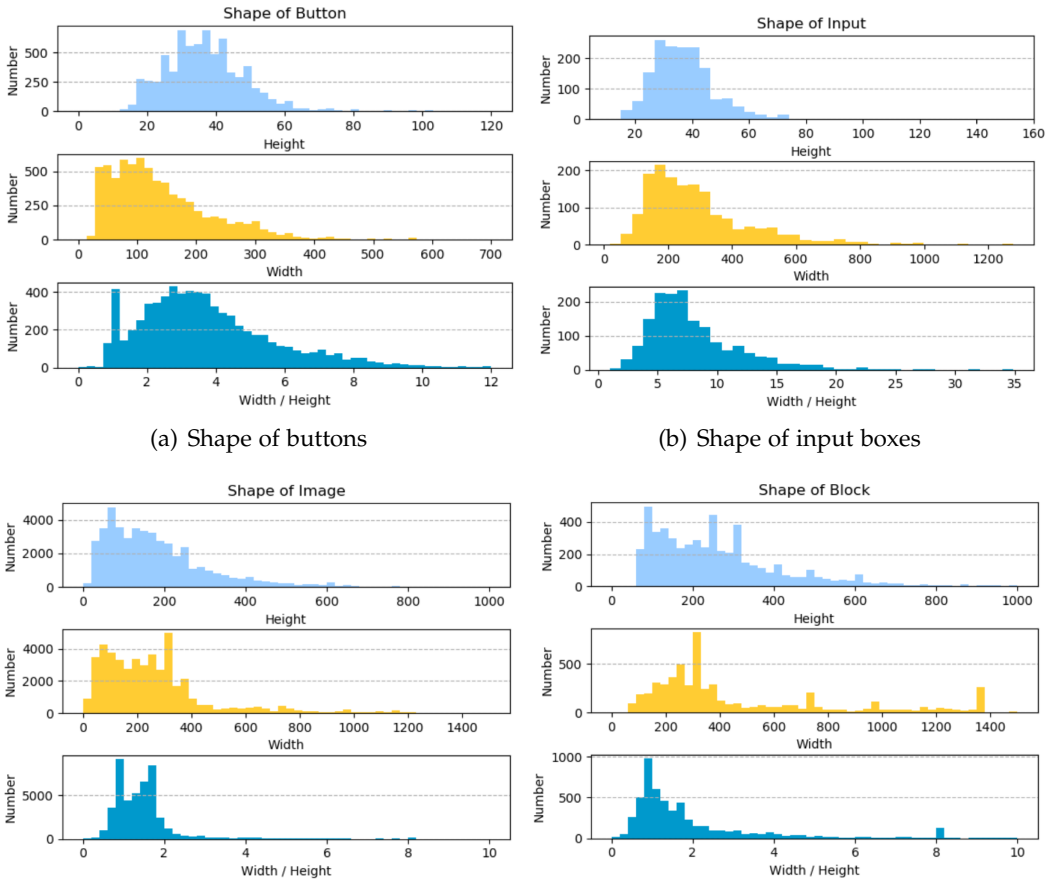
Figure 4.9: The demonstrations of a block. Blocks are drawn with green bounding box, and they usually are bordered regions where contain multiple components.

4.3.5 Irregular Shaped Components Selection

On the ground of the statistics 4.8, 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, 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 show in Figure 4.10, 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 4.10: 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 in specific ranges. For example, most of buttons' height is in the range of 20 pixels to 60 pixels, and most of their width are between 40 pixels and 300 pixels, the major aspect ratio of buttons is on a scale of one to eight.

Therefore, adhoc filters is built according to the heuristics 4.1. Four rules are defined here and the filters are scattered over the whole process when implemented.

The *SmallComponent* rule leave 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 4.1: Heuristics based on the statistics.

4.3.6 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.

Observation about such nested components indicates that it is impossible to view a button inside another button or a 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 4.11.

As elaborated in the pre-processing section 4.2, 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 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.

Figure 4.12 shows the binary map that integrates some buttons into the background and identify them as an entire image component. But we can observe form the binary map 4.12(a) that a black hole in 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.

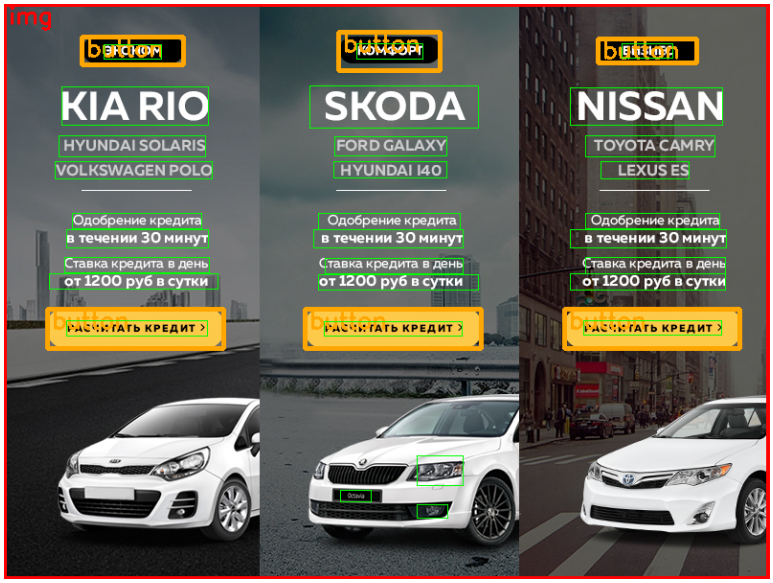


Figure 4.11: Example of figure that some interactive elements on a complicated image background

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.

4.4 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 4.2.

4.4.1 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 real interface, and I define six classes according to the related objects and tags that perform particular functionalities in the front-end languages.

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

(a) Binary map (b) Opposite map

Figure 4.12: The demonstration of the binary map and its opposite image.

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

Table 4.2: The categories and classes of UI components.

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 in stead 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 tiny image but it exists independently in Android development, so I separate this class from image. Although those resource can sometimes be implemented as functional elements, typically as hyperlinks, I still treat them as 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 graphic interface consisting of various components. The meaning of this category is to find out some obvious hierarchies and layers for future code generation.

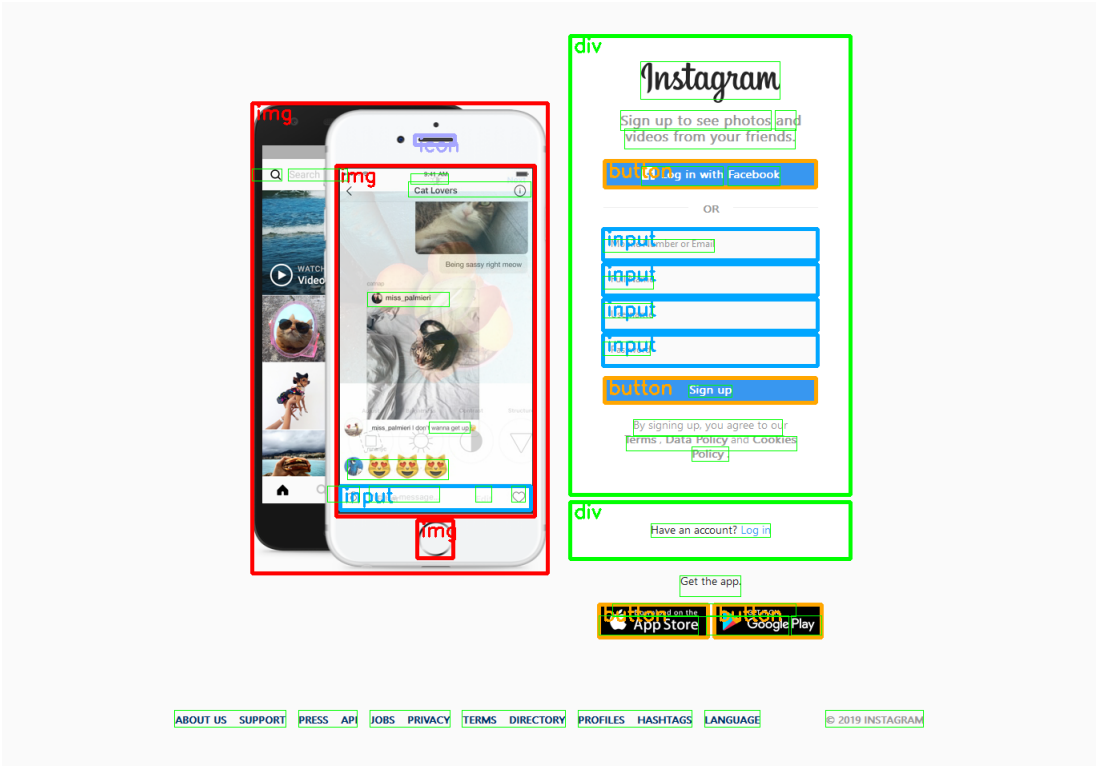


Figure 4.13: Demo of a labeled web page screenshot. Various classes are tagged with different colours of bounding box; the slim green boxes in this picture are the results from the CTPN showing the text recognition.

The Figure 4.13 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 segmented into components with tags in which we know the locations and classes of those detected objects.

4.4.2 Classifier Model

To perform components classification, I recur to machine learning methods. Several techniques have been tried in the process including Super Vector Machine classifier 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 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), the Histogram of Oriented Gradients (HOG) combined with Support Vector Machine (SVM) and the Convolutional Neural Network (CNN).

Again, this part does not elaborate the technical details of those techniques, instead, it just introduces the basic theories of them and state the experimental results.

4.4.2.1 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. This pipeline is similar to the SIFT to a degree, but it has some unique advantages compared to the former. 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.

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, the gradient of this image is calculated at the beginning. Basically, the way to compute the gradient is same as formulae 4.3, but the author implement it by the following filter kernel:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{4.4}$$

$$g(x,y) = w * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(x,t) f(x-s, y-t) \tag{4.5}$$

The kernel 4.4 is convoluted on image to compute the gradinet, the general expression of a convolution is stated in 4.5, 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 depend 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:

$$f = \frac{v}{\sqrt{\|v\|_2^2 + e^2}} \tag{4.6}$$

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), a supervised learning models based on learning algorithm to perform classification and regression.

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.

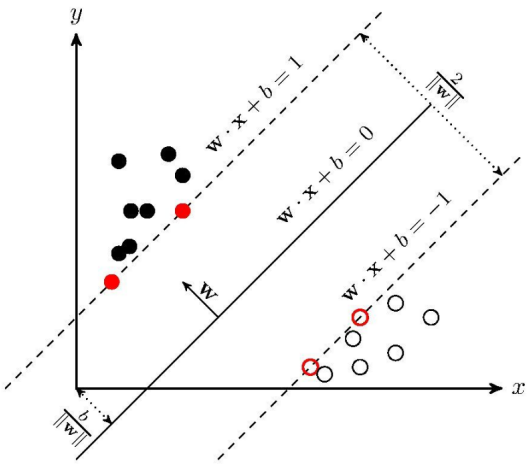


Figure 4.14: Hyperline partitioning two groups of points

Figure 4.14 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.

4.4.2.2 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 objection recognition and action recognition, 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. It search for the extreme points in the space and extracts their location, scale and orientation, and then all those features are stored in database through 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 Eucidean distance of their feature vectors. Mainly four steps involved in SIFT algorithm, this section is just a brief summary of it.

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 are obtained as the difference between two images blurred by different Gaussian filters with different σ, and this process is done for all layers in Gaussian Pyramid as illustrated below.

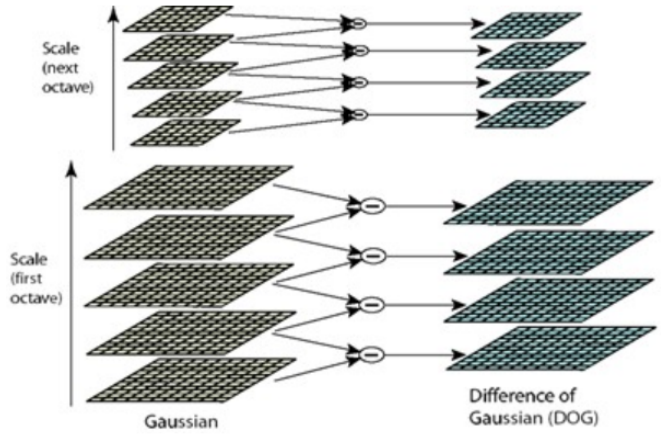


Figure 4.15: DoG are computed in all layers in Gaussian Pyramid

After the DoG are 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 potential key point representing the feature in its scale.

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

Orientation assignment: The keypoints are assigned orientation in this step by calculating the gradients with its neighbours on this scale. The gradients' direction and magnitude is 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.

4.4.2.3 CNN

Compared with aforementioned techniques, the Convolutional Neural Network (CNN) is a more end-to-end method. 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.

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 variant. 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 locate in the center 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 srtucture is presented in Figure 4.16. In order to offset the effect caused by the insufficient parameters in a shallow network, each layer is created more broad.

4.4.3 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 image of web and mobile application. Although there are some slight variations among interface elements of web and mobile apps, those nuance does impose considerable influence on the model's performance. Therefore, all compo-

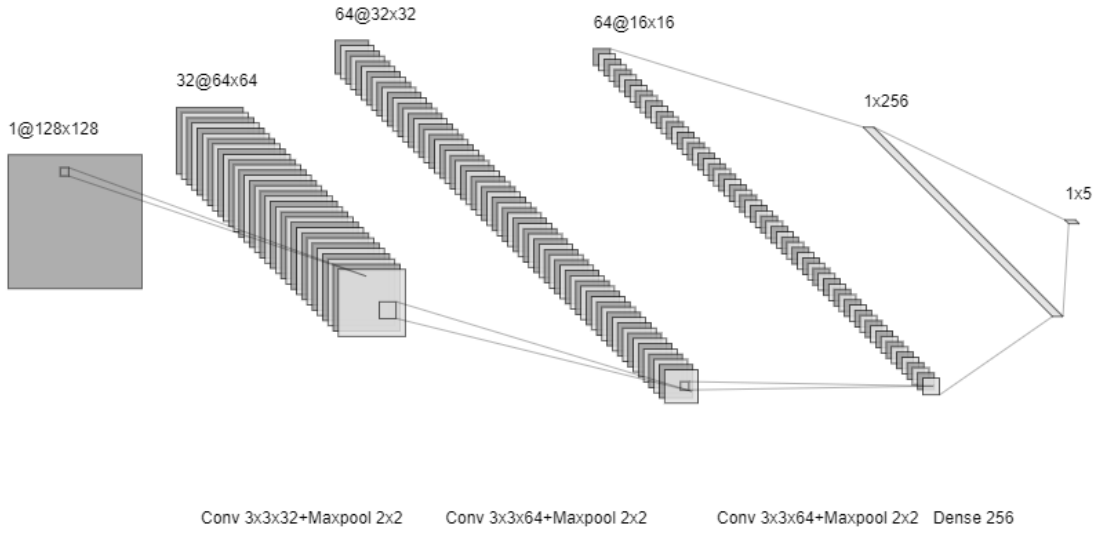


Figure 4.16: The structure of the four-layer network. A 3x3 sliding window is adopted to move through the original image that is resized into size of 128x128. All convolutional layers are followed by a 2x2 max-pooling layer. The output layer is a five-class softmax to classify the input into *image, button, input box, icon* and *text*.

nents in the same class from different sources are collected together and fed into the training models.

4.5 Text Processing

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

4.5.1 Introduction

The CTPN refers to the state-of-the-art object detection methods, especially the Faster RCNN, but the authors proposed some novel improvement to adapt the network to the natural scene text line detection based on the visual properties of 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. For instance, anchor regression that widely used in recent object detection methods is 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 a significant progress to the localization accuracy. This novelty departs from the Region Proposal Network in the FRCNN, which attempts to predict a whole object, hence is difficult to provide a satisfied localization accuracy in the case of text detection.

Second, in view of the consecutiveness of 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 the 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 [8] on the ICDAR 2013, and 0.61 F-measure over 0.54 in [35] on the ICDAR2015).

4.5.2 Technical Details

Three critical novelties are proposed in the CTPN: detecting in fine-scale proposals, recurrent connectionist text proposals and side-refinement. Figure 4.17 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 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 text line is that it is a sequential region where it does not have an obvious 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, focuses on the text line and region level, so the accuracy might not be satisfied if it still treat the targets as single objects.

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

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

predicted y-coordinates that represents the height of this anchor for k anchors of each window.

Recurrent Connectioist Text Proposals: The fine-scale proposals results 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 text. Furthermore, the CTPN integrates the context information into the convolutional layer.

To this end, the CTPN, the long short-term memory (LSTM) is used for the recurrent neural network (RNN) layer. The authors adopted some tricks to address vanishing gradient problem. Moreover, they use a bi-directional LSTM 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.

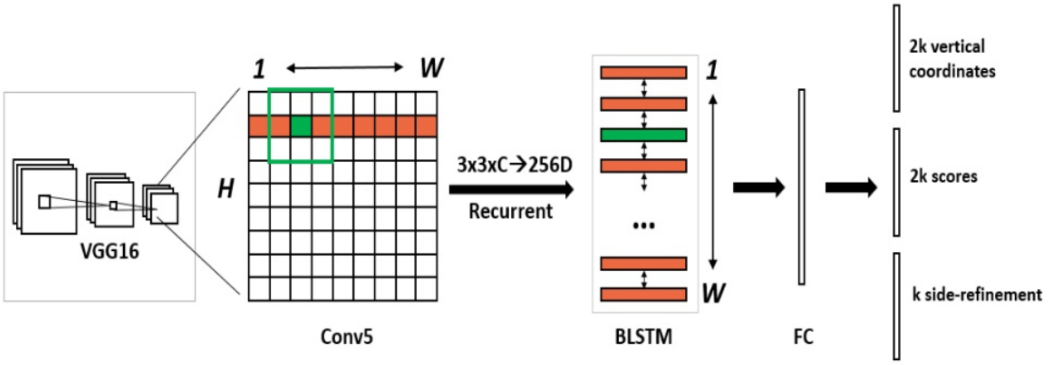


Figure 4.17: The overall structure of the Connectionist Text Proposal Network (CTPN). A 3x3 sliding window is applied through the last convulotional map of the base network (VGG16). Then the sequence of windows in each row are recurrently connected by a Bio-directional LSTM to gather the sequential context information. At 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.

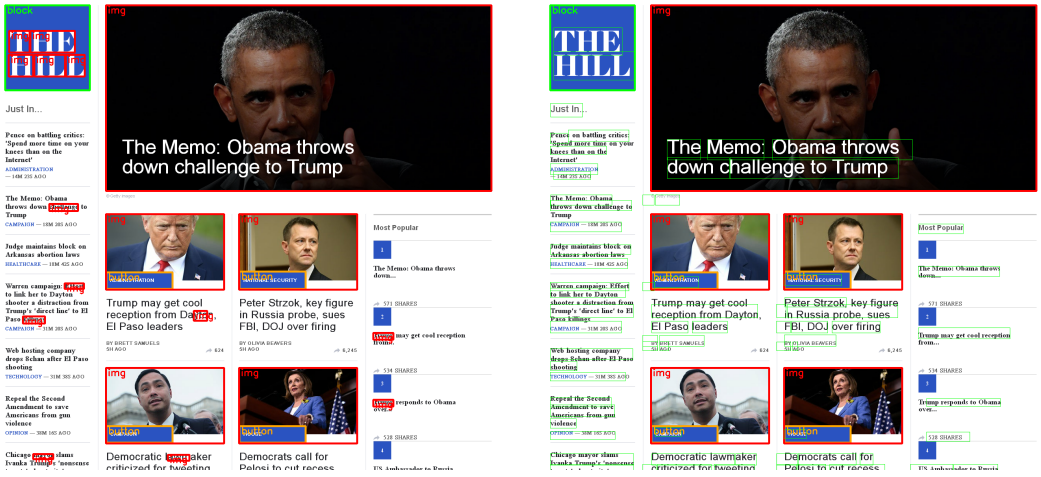
4.6 Merge

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 4.1. 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 source, 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 the 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 4.18.



(a) Result of the UI components detection

(b) Result after merging with text detection

Figure 4.18: A section of web application interface. The 4.18(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 4.18(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.

To achieve this, the resulting UI components on the left hand side are filtered by computing the overlap with the text lines. This is based on 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:

$$\forall j \in Text(IoA(i,j) < 0.7) \tag{4.7}$$

$$(\sum_{j \in Text} Inter(j,i))/Area_i < 0.85 \tag{4.8}$$

The requirement 4.7 means the intersection area of a single text region with the component i should smaller than the 70 percentage of component i's area; and the expression 4.8 stipulates that the total text area of an interface element should less than 85 percentage of the its area.

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

Chapter 5

Why Not Deep Learning?

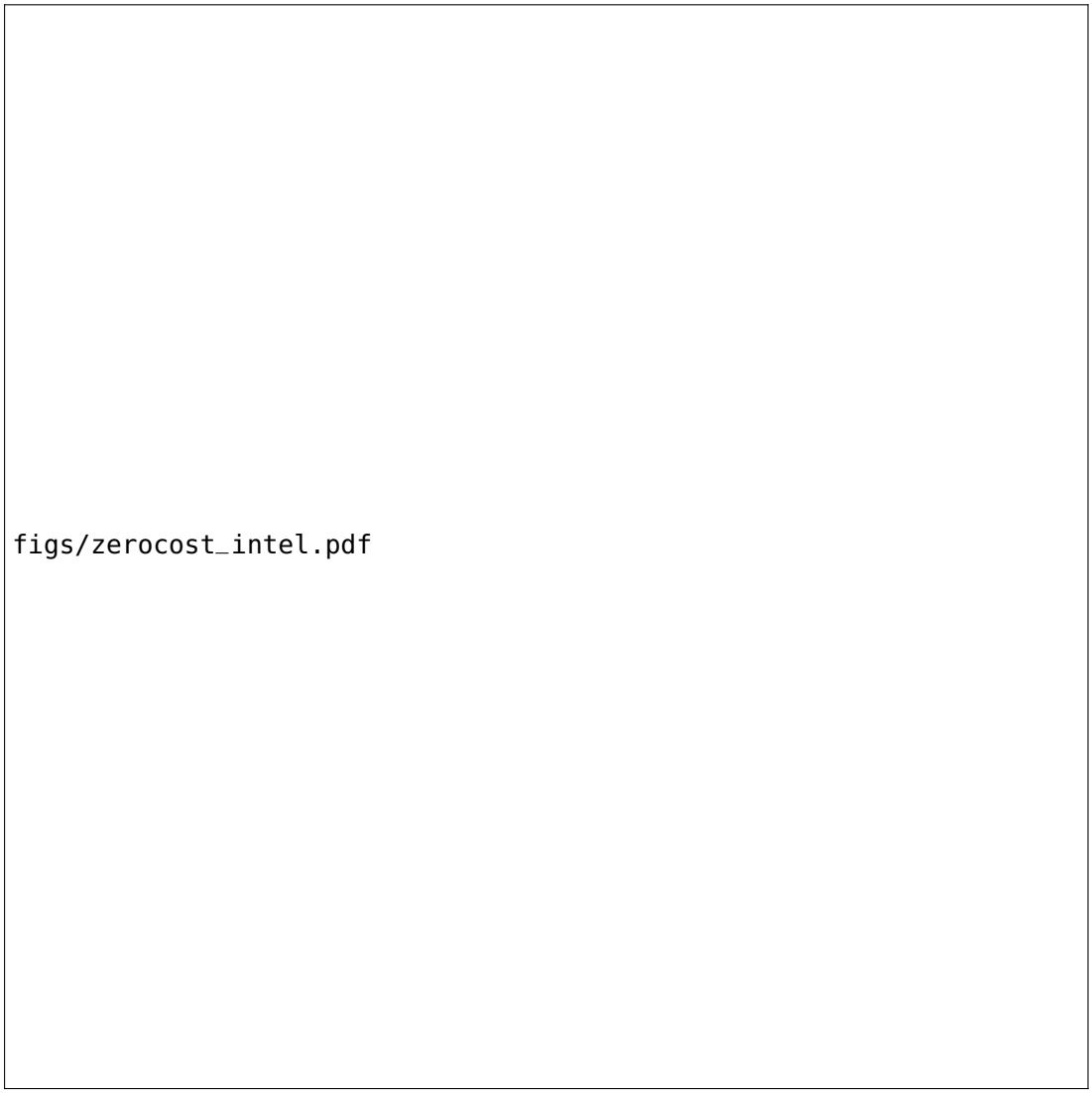
Chapter 6

Results

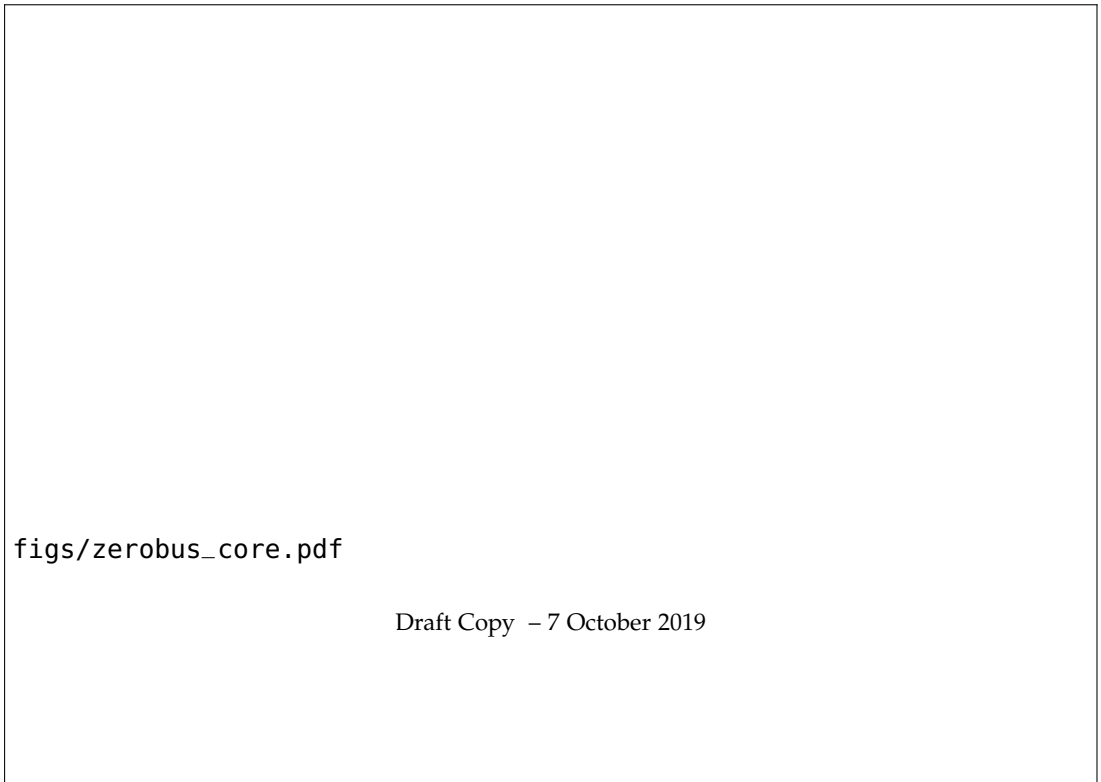
6.1 Direct Cost

Here is the example to show how to include a figure. Figure 6.1 includes two subfigures (Figure 6.1(a), and Figure 6.1(b));

6.2 Summary



(a) Fraction of cycles spent on zeroing



Chapter 7

Conclusion

Summary your thesis and discuss what you are going to do in the future in Section 7.1.

7.1 Future Work

Good luck.

Bibliography

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi:10.1145/358141.358147. (cited on page 3)

Moon, D. A., 1984. Garbage collection in a large LISP system. In LFP '84: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA, Aug. 1984), 235–246. ACM, New York, New York, USA. doi:10.1145/800055.802040. (cited on page 3)

Ungar, D., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In SDE 1: *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, USA, Apr. 1984), 157–167. ACM, New York, New York, USA. doi:10.1145/800020.808261. (cited on page 3)