School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

APPM4058A Project - Barcode Extraction

Wesley Earl Stander - 1056114 - Coms Hons

June 6, 2020

1 Problem Domain

Detecting bar-codes on the back of books is a relatively easy task for any cashier when scanning up a book for sale. The task can become quite complex when it comes to detecting bar-codes from an image on a computer. The task becomes even more complex when the detector must detect images that are both gray-scale or colour. The requirements of this problem require a bar-code to be detected utilizing spatial features and not spectral features. A bar-code consists of a set of varying width vertical bars of the same vertical length. The bar-code often has an ISBN number above it and another number below it. The solution must be a binary image of the bar-code and hence must be able to detect the distinct bars in images of lower quality. The image produced by the code used in this project must produce distinct bars that can be scanned in a binary image format.

2 Data-set

A set of pictures were sourced off the internet alongside pictures taken with my phone. No data-set was available with backs of books so the images had to be sourced individually. Some images were lower quality that causes a rectification step to sharpen the image so that bar information could be extracted. The non-uniform lighting with respect to photos taken with my phone did not allow for those photos to work with this method.

3 Solution Approach

The image must be converted to gray-scale if it is in colour so that the spatial information is the only information being considered. The bar-code if upright in the image with the vertical bars parallel to the y-axis or closer to the y-axis than the x-axis provides a section of spatial information that has high horizontal gradients and low vertical gradients. The technique in the DIP notes makes use of a Sobel operator in the respective directions to detect regions of high gradient in those directions so the image is processed with both vertical and horizontal Sobel operators to produce two results. To extract the regions that have a low vertical gradient and high horizontal gradient the y-gradient of the Sobel gradient is subtracted from the x-gradient of the Sobel gradient. This result requires the absolute to be taken so that the background information becomes black and the foreground information white instead of varying shades of gray.

```
imgG = skimage.color.rgb2gray(img) #convert colour image to gray-scale if colour
imgV = skimage.filters.sobel_v(imgG) #vertical gradient image
imgH = skimage.filters.sobel_h(imgG) #horizontal gradient image
imgGradient = numpy.abs(imgH- imgV) #image with high horizontal gradient and low vertical gradient
```

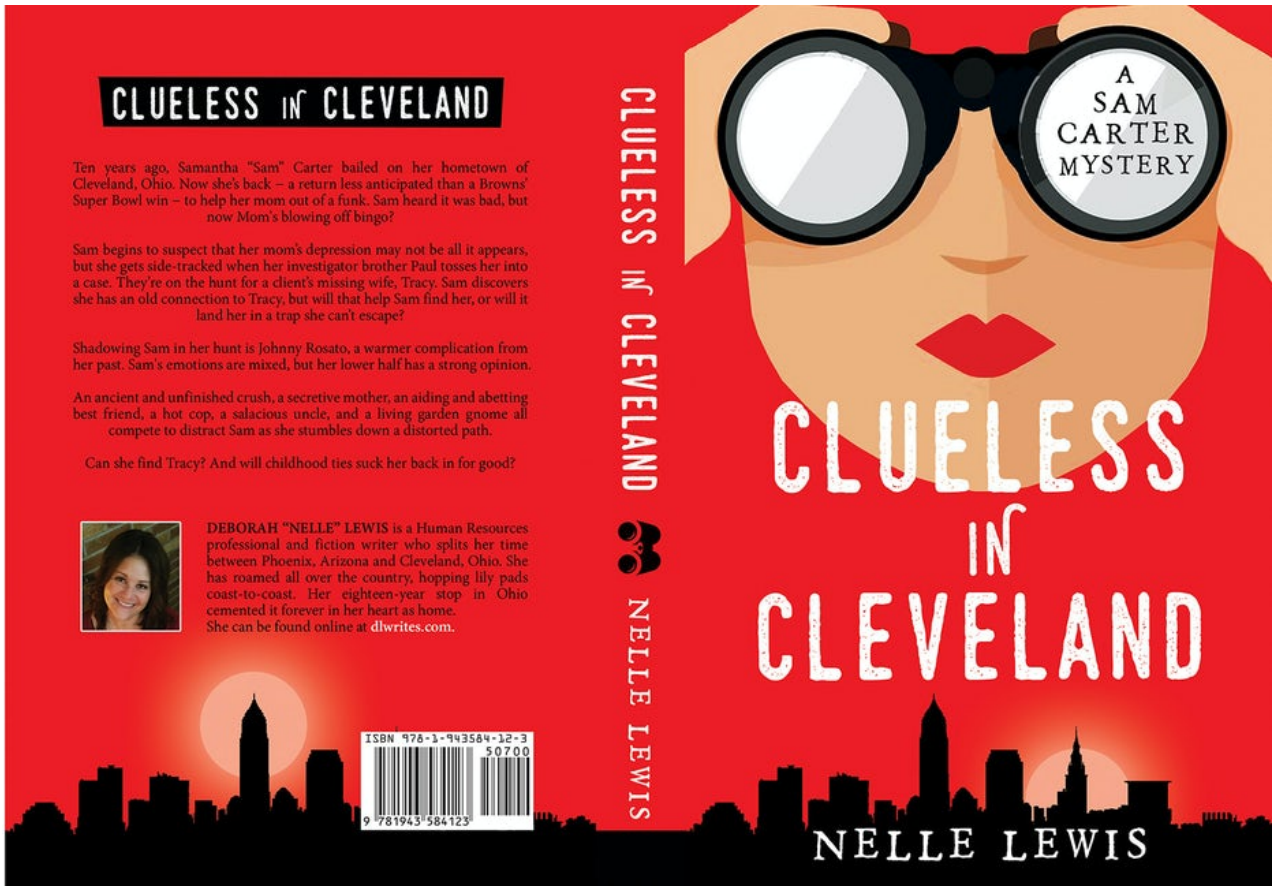

Figure 1: Example input

The next step is to blur out the details of the small text on the bar-code alongside the bars in order to extract the block where the bar-code resides. After blurring with a large box filter for the size of the image, at-least $1/100^{th}$ of the largest dimension of the image. This blurring causes the bar-code block to become mostly white. The image now has a foreground set of pixels that are closer to white and background pixels which are closer to black. The bar-code block is mostly white and by utilizing an otsu-thresholding on the gradient image, the foreground pixels which include the bar-code can be extracted on the blurred image.

```
filterSize = (int)(max(img.shape[0], img.shape[1])/100) #get approximate filter size
#blur image with box filter and zero padding
blurred = skimage.exposure.rescale_intensity(scipy.signal.convolve2d(imgGradient,
    skimage.morphology.square(filterSize)/filterSize**2, mode="same"))
n = skimage.filters.threshold_otsu(imgGradient) #obtain otsu-thresholding value from gradient
bar = skimage.exposure.rescale_intensity(blurred) >= n #perform otsu thresholding
```

The next step involves closing the bar-code if there are any sections that still have gaps. The entire image is processed with a closing using a structuring element of size 5x5 to close the gaps. Now the entire block where the bar-code is has been filled but so has all foreground pixels throughout the image. There is a dual property about the rectangle that can be exploited to extract only the rectangle. The rectangle has both a wide foreground region alongside a slightly less tall vertical region if sliced along both planes. The ratio of a bar-code width to it's height is approximately 1.625x. To create a narrow bar structuring element to erode along the x-axis with the minimum height is 1. The bar needs to be much wider than tall and also not too long to not include all images. A width of 21 was chosen experimentally with a set of

images. The image is then eroded 4 times with the bar then dilated the same amount of times to remove all features that don't contain horizontal blocks. The same occurs in the vertical direction. A bar structuring element of height 13 was selected as it is 1.625x smaller than 21 and a the smallest width of 1 is chosen. The image is eroded 4 times then dilated 4 times with the vertical bar structuring element.

```
SE = numpy.ones((5,5))
b = m.closing(bar, SE) #close any gaps in foreground

count = 4
#removing non wide elements
for i in range(count):
    b = m.erosion(b, numpy.ones((1,21)))
for i in range(count):
    b = m.dilation(b, numpy.ones((1,21)))
#removing non tall elements
for i in range(count):
    b = m.erosion(b, numpy.ones((13,1)))
for i in range(count):
    b = m.dilation(b, numpy.ones((13,1)))
```

At the end of this set of code b is an image that contains only the section where the bar-code is as foreground pixels displayed in figure 2. The section doesn't contain the full bar-code so we need to extract a rectangle surrounding the foreground pixels. After the bound of the foreground pixels in the 4 cardinal directions are acquired then the rectangle can be formed in an image to mask the original image. The central pixel is acquired and dilated in the masking image by the rectangle structuring element to achieve the mask. The code below produces Figure 3.

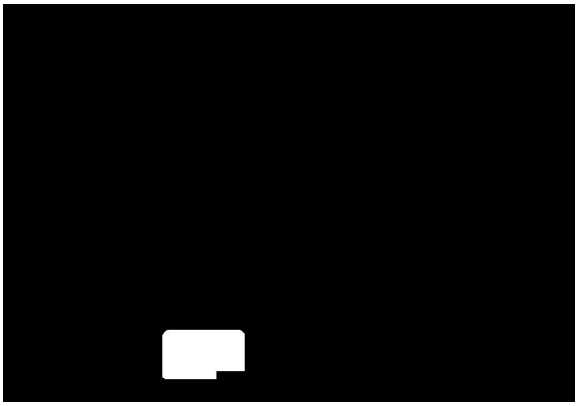

Figure 2: Bar-Code location

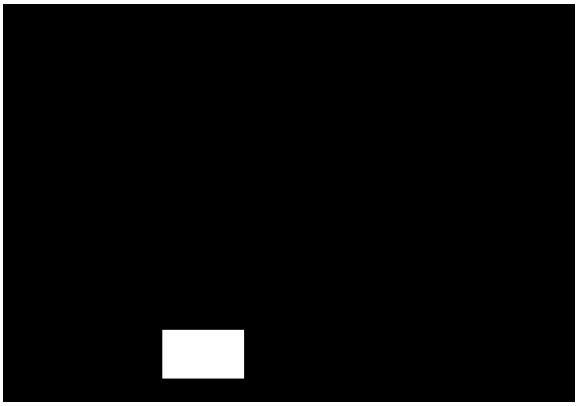

Figure 3: Full bar-code Location

```
def ExtractBounds(I):
    positions = numpy.nonzero(I)
    top = positions[0].min()
    bottom = positions[0].max()
    left = positions[1].min()
    right = positions[1].max()
    width = right - left
    height = bottom - top
    x = (int)(right - width/2)
    y = (int)(bottom - height/2)
    s = numpy.ones((height, width)) #structuring element
    h = numpy.zeros((b.shape[0], b.shape[1])) #black image
```

```
    print(top, bottom, left, right)
    h[y][x] = 1 #bar-code location
    detect = skimage.morphology.dilation(h, s) #bar-code mask placement
    return detect, [top,bottom], [left,right]
```

The image is masked by multiplying in the mask and sliced to acquire the pixels that are only the bar-code pixels. Now that the bar-code has been extracted the white section can be bordered by the some fluff pixels caused by the earlier blurring. The bar-code image must undergo an unsharp masking in case the quality of the image is low and then an otsu-thresholding to separate the foreground from the background. This process of extracting bounds is repeated on the thresholded bar-code image and this extracts only the white section of the bar-code. The exact bar-code has been extracted now.

```
d = ExtractBounds(b)
imgOut = d[0] * rgb2gray(img)
barcodeOnly = imgOut[d[1][0]:d[1][1], d[2][0]:d[2][1]]
br = skimage.filters.unsharp_mask(barcodeOnly)
T = skimage.filters.threshold_otsu(br)
br = br >= T #extract main bar-code only
dd = ExtractBounds(br) #extract white only
final = barcodeOnly[dd[1][0]:dd[1][1],dd[2][0]:dd[2][1]]
```

The final step requires turning the image into a binary image showing only the bar-code. For lower quality images the bar-code must undergo multiple unsharp masking to improve the representation of the bars as they often blur if they are close together. For some images they have to undergo up to 5 unsharp maskings to allow the otsu-thresholding to correctly extract the bars. The technique to extract the bars correctly requires a dilation of the white space with large vertical bars the height of half of the vertical height of the bar-code image. The dilation is corrected by an erosion afterwards. This removes any noise in the image that is not bar information. The image then undergoes 4 unsharp maskings to reveal any gradient information as actual bars. This allows for an otsu-thresholding to separate the background (bars) and the foreground (white-space).

```
final = skimage.morphology.dilation(final, numpy.ones(((int)(final.shape[0]/2),1)))
final = skimage.morphology.erosion(final, numpy.ones(((int)(final.shape[0]/2),1)))
for i in range(4):
    final = skimage.filters.unsharp_mask(final)
TH = f.threshold_otsu(final)
output = final > TH
```

Figure 4: Bar-code outputs

The final image presented by this code produces a crisp binary image that can be scanned by a bar-code scanner. The issues with anti-aliasing blurring bars has been corrected by this last section of the code. The final images include an

image of the original bar-code, a binary image of the bars improved through sharpening and bar extraction, as well as a binary image which is an overlay of the sharpened bars as well as sharpened text visible around the bars. The bars only image is the preferred image for scanning the bar-code.

4 Source Code list

4.1 Libraries

1. numpy
2. matplotlib.pyplot
3. skimage.color.rgb2gray
4. skimage.morphology
5. skimage.filters
6. skimage.exposure.rescale_intensity
7. scipy.signal.convolve2d

5 Further examples

Figure 5: Synanon back cover

Figure 6: Synanon Extracted bar-code

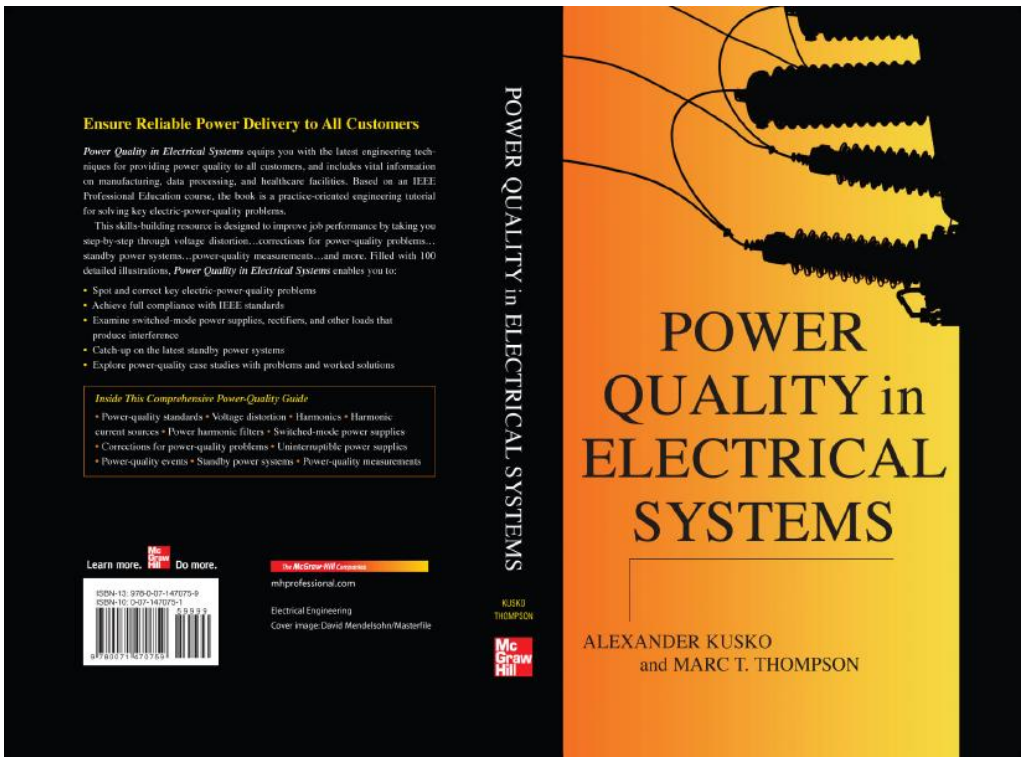

Figure 7: Power back cover

Figure 8: Power Extracted bar-code