Adaptive Thresholding using Integral Image
170050070,170070007

Objective

Image thresholding is a common task in many computer vision and graphics applications. The goal of thresholding an image is to classify pixels as either "dark" or "light". Adaptive thresholding is a form of thresholding that takes into account spatial variations in illumination. In this project we do it with the help of integral image.

Algorithm

Integral Image

To compute the integral image, we store at each location, I(x, y), the sum of all f (x, y) terms to the left and above the pixel (x, y). This is accomplished in linear time using the following equation for each pixel (taking into account the border cases).

$$I(x, y) = f(x, y) + I(x-1, y) + I(x, y-1) - I(x-1, y-1)$$

Once we have the integral image, the sum of the function for any rectangle with upper left corner (x1 , y1), and lower right corner (x2 , y2) can be computed in constant time using the following equation

$$\sum_{x=x1}^{x2} \sum_{y=y1}^{y2} f(x, y) = I(x2, y2) - I(x2, y1-1) - I(x1-1, y2) + I(x1-1, y1-1)$$

Also this is done by the function integral(matrx) in cv2.

Algorithm

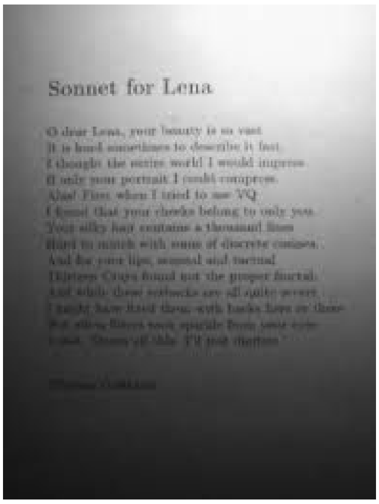
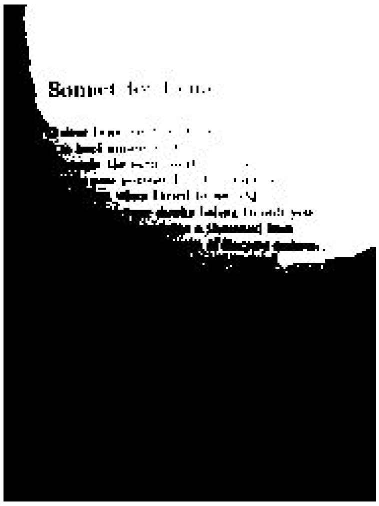
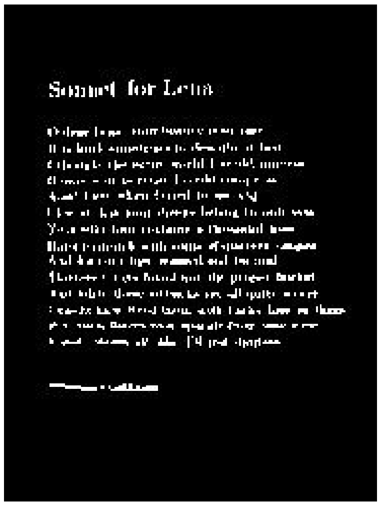
In the first pass through the image we calculate its integral image. Then in the second pass we consider a size s*s square (where s is determined by testing on various values) centred on the pixel we are considering. We calculate the average value of pixels in this s*s square in O(1) time because of its integral properties. If the value of the current pixel is t percent less than this average then set it to black otherwise set it to white (here t is the threshold value). The pseudo -algorithm is as given below where w an h are the image width and height respectively.

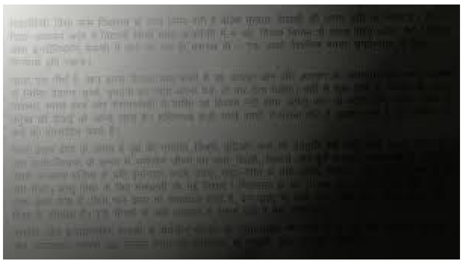
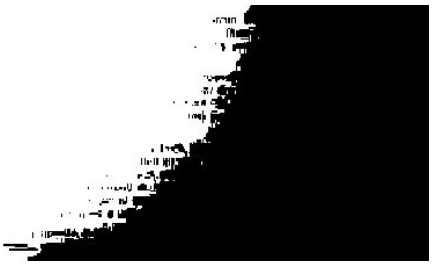
for $i = 0$ to w **do**
    for $j = 0$ to h **do**
        $x1 \leftarrow i - s/2$ {border checking is not shown}
        $x2 \leftarrow i + s/2$
        $y1 \leftarrow j - s/2$
        $y2 \leftarrow j + s/2$
        $count \leftarrow (x2 - x1) \times (y2 - y1)$
        $sum \leftarrow intImg[x2, y2] - intImg[x2, y1-1] - intImg[x1-1, y2] + intImg[x1-1, y1-1]$
        if $(in[i, j] \times count) \leq (sum \times (100 - t)/100)$ **then**
            $out[i, j] \leftarrow 0$
        else
            $out[i, j] \leftarrow 255$
        end if
    end for
end for

.

Experiments

Although the algorithm tends to work for every image, it usually does better for images which have variations in illumination (this is because adaptive thresholding thresholds in a local area whereas Otsu's method looks globally) . We have provided a comparison between OTSU's thresholding algorithm and our adaptive thresholding.

		
Original Image	Otsu's thresholding	Adaptive thresholding

		
Original Image	Otsu’s thresholding	Adaptive thresholding

		
Original Image	Otsu’s thresholding	Adaptive thresholding

As can be seen clearly in the above images the Adaptive thresholding does much better than OTSU’s method.

Code

```
import cv2
import math
import numpy as np
import time
import sys

def thresholdIntegral1(inputMat,s,T = 0.15):
    # outputMat=np.uint8(np.ones(inputMat.shape)*255)
    outputMat=np.zeros(inputMat.shape)
    nRows = inputMat.shape[0]
    nCols = inputMat.shape[1]
    S = int(max(nRows, nCols) / 8)

    s2 = int(S / 4)
```

```python
    for i in range(nRows):
        y1 = i - s2
        y2 = i + s2

        if (y1 < 0) :
            y1 = 0
        if (y2 >= nRows):
            y2 = nRows - 1

        for j in range(nCols):
            x1 = j - s2
            x2 = j + s2

            if (x1 < 0) :
                x1 = 0
            if (x2 >= nCols):
                x2 = nCols - 1
            count = (x2 - x1)*(y2 - y1)

            sum=s[y2][x2]-s[y2][x1]-s[y1][x2]+s[y1][x1]

            if ((int)(inputMat[i][j] * count) < (int)(sum*(1.0 -
T))):
                outputMat[i][j] = 255
                # print(i,j)
            # else:
            #     outputMat[j][i] = 0
    return outputMat



if __name__ == '__main__':
    ratio=1
    image = cv2.imdecode(np.fromfile(sys.argv[1], dtype=np.uint8),
0)
    img = cv2.resize(image, (int(image.shape[1] / ratio),
int(image.shape[0] / ratio)), cv2.INTER_NEAREST)

    retval, otsu = cv2.threshold(image, 0, 255, cv2.THRESH_OTSU)
    cv2.namedWindow('OTSU threshold',0)
```

```
    cv2.imshow('OTSU threshold',otsu)
    cv2.imwrite('otsu_results.jpg',otsu)

    thresh = cv2.adaptiveThreshold(img,  255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 9, 9)
    retval, thresh = cv2.threshold(img, 150, 255, cv2.THRESH_OTSU)
    retval, thresh = cv2.threshold(img, retval, 255,
cv2.THRESH_OTSU)




    roii = cv2.integral(img)

    for j in range(1):
        thresh = thresholdIntegral1(img, roii)
    cv2.namedWindow('fast inergral threshold',0)
    cv2.imshow('fast inergral threshold',thresh)
    cv2.imwrite('results.jpg', np.uint8(thresh))


    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

References

1)Adaptive Thresholding using the Integral Image