A Threshold Selection Method from Gray-Level Histograms using the Otsu method

2.11.2020

PRATEEK SAHU

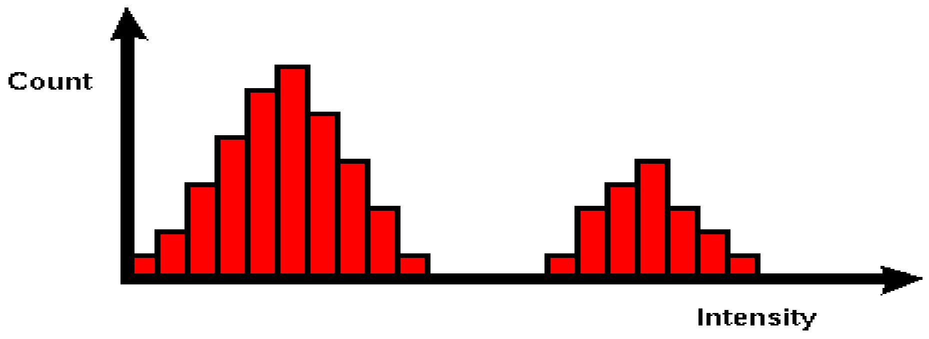
171CO130

Overview

Segmentation involves separating an image into regions (or their contours) corresponding to objects. We usually try to segment regions by identifying common properties. Or, similarly, we identify contours by identifying differences between regions (edges).The simplest property that pixels in a region can share is intensity. So, a natural way to segment such regions is through thresholding, the separation of light and dark regions.

Image thresholding is a simple, yet effective, way of partitioning an image into a foreground and background. This image analysis technique is a type of image segmentation that isolates objects by converting grayscale images into binary images.

In an image processing context, the histogram of an image normally refers to a histogram of the pixel intensity values. This histogram is a graph showing the number of pixels in an image at each different intensity value found in that image. For an 8-bit grayscale image there are 256 different possible intensities, and so the histogram will graphically display 256 numbers showing the distribution of pixels amongst those grayscale values.



Histogram of grayscale image.

Problem Statement

To set a global threshold we usually look at the histogram to see if we can find two or more distinct modes—one for the foreground and one for the background. **A histogram is a probability distribution.**

$$p(g) = n_g/n$$

That is, the number of pixels n_g having grayscale intensity g as a fraction of the total number of pixels n.

One extremely simple way to find a suitable threshold is to find each of the modes (local maxima) and then find the valley (minimum) between them. While this method appears simple, there are two main problems with it:

1. The histogram may be noisy, thus causing many local minima and maxima. To get around this, the histogram is usually smoothed before trying to find separate modes.
2. The sum of two separate distributions, each with their own mode, may not produce a distribution with two distinct modes.

Another way to look at the problem is that we have two groups of pixels, one with one range of values and one with another. What makes thresholding difficult is that these ranges usually overlap. What we want to do is to minimize the error of classifying a background pixel as a foreground one or vice versa. To do this, we try to minimize the area under the histogram for one region that lies on the other region's side of the threshold. The problem is that we don't have the histograms for each region, only the histogram for the combined regions.

Solution

A data-driven way which can adaptively find the optimal threshold to distinguish two-class data — Otsu thresholding. This method can be applied in image segmentation and image binarization. **It is based on minimizing the variance of each cluster which we desire to separate.**

Consider that we have an image with L grey levels and its normalized histogram (Probability Histogram) i.e., for each grey-level value I, P(i) is the normalized frequency of i.

Assuming that we have set the threshold at T, the -normalized- fraction of pixels that will be classified as background and object will be:

$$q_b(T) = \sum_{i=1}^{T} P(i), \qquad q_o(T) = \sum_{i=T+1}^{L} P(i) \quad (q_b(T) + q_o(T) = 1)$$

The mean grey-level value of the background and the object pixels will be:

$$\mu_b(T) = \frac{\sum_{i=1}^{T} iP(i)}{\sum_{i=1}^{T} P(i)} = \frac{1}{q_b(T)} \sum_{i=1}^{T} iP(i) \qquad \mu_o(T) = \frac{\sum_{i=T+1}^{L} iP(i)}{\sum_{i=T+1}^{L} P(i)} = \frac{1}{q_o(T)} \sum_{i=T+1}^{L} iP(i)$$

The mean grey-level value over the whole image (grand mean) is:

$$\mu = \frac{\sum_{i=1}^{L} iP(i)}{\sum_{i=1}^{L} P(i)} = \sum_{i=1}^{L} iP(i)$$

The variance of the background and the object pixels will be:

$$\sigma_b^2(T) = \frac{\sum_{i=1}^{T}(i-\mu_b)^2 P(i)}{\sum_{i=1}^{T} P(i)} = \frac{1}{q_b(T)} \sum_{i=1}^{T}(i-\mu_b)^2 P(i)$$

$$\sigma_o^2(T) = \frac{\sum_{i=T+1}^{L}(i-\mu_o)^2 P(i)}{\sum_{i=T+1}^{L} P(i)} = \frac{1}{q_o(T)} \sum_{i=T+1}^{L}(i-\mu_o)^2 P(i)$$

The variance of the whole image is:

$$\sigma^2 = \sum_{i=1}^{L}(i-\mu)^2 P(i)$$

Within-class and between-class variance:

It can be shown that the variance can be written as follows:

$$\sigma^2 = q_b(T)\sigma_b^2(T) + q_o(T)\sigma_o^2(T) \quad + \quad q_b(T)(\mu_b(T)-\mu)^2 + q_o(T)(\mu_o(T)-\mu)^2 =$$

$$\sigma_W^2(T) + \sigma_B^2(T)$$

where $\sigma_W^2(T)$ is defined to be the within-class variance and $\sigma_B^2(T)$ is defined to be the between-class variance.

Determining the threshold

Since the total variance does not depend on t, the t minimizing within-class variance will be the t maximizing Between class variance.

Let's consider maximizing Between class variance, we can rewrite Between class variance as follows:

$$\sigma_B^2 = \frac{[\mu(T) - \mu q_B(T)]^2}{q_B(T) q_o(T)}$$

Where:

$$\mu(T) = \sum_{i=1}^{T} iP(i)$$

Algorithm goes as follows:

a) we start from the beginning of the histogram and test each grey-level value and find Between class variance and Within class variance
b) We choose the threshold T that maximizes Between class variance or Minimizes Within class variance so that classes are as close as possible.
c) Then we use this t for image binarization all pixels greater than T are tuned white and below t are turned black.
d) We output the binarized image.

Implementation

I have Implemented The Algorithm in Python Following is code Snippet:

Full Implementation of Otsu ,results and images can Also be found in following link:

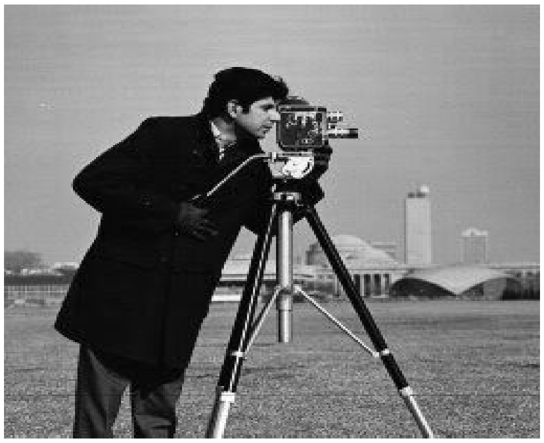
https://github.com/Prateek1337/Otsu_Thresholding

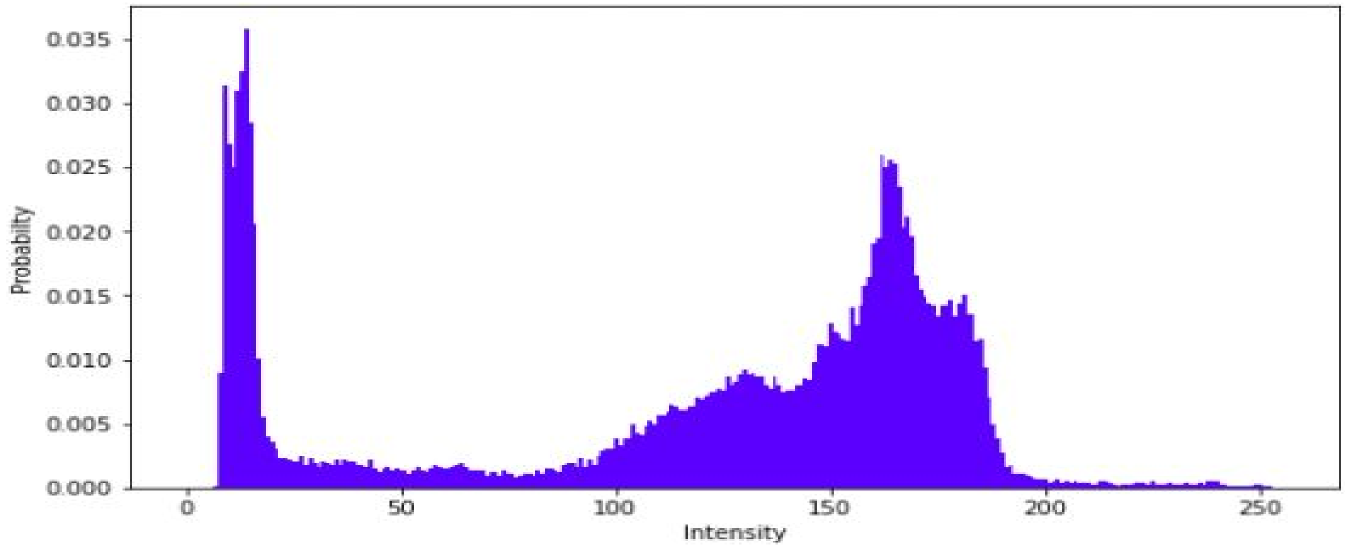
```
import math
import numpy as np
from matplotlib import pyplot as plt
from PIL import Image

def Hist(img):# This will calculate the probability histogram of the image
    row, col = img.shape
    y = np.zeros(256)
    for i in range(0,row):
        for j in range(0,col):
            y[img[i,j]] += 1
    cnt=countPixel(y)
    y=y/cnt
    x = np.arange(0,256)
    f, axarr = plt.subplots(1,2,figsize=(20,5))
    axarr[0].bar(x, y, color='b', width=1, align='center')
    axarr[0].set_ylabel("Probabilty")
    axarr[0].set_xlabel("Intensity")
    plt.show()
    return y

def regenerate_img(img, threshold): # This used for image Thresholding
when we know a threshold.
    row, col = img.shape
    y = np.zeros((row, col))
    for i in range(0,row):
        for j in range(0,col):

            if img[i,j] >= threshold:

                y[i,j] = 255

            else:

                y[i,j] = 0

    return y




def countPixel(h): # This will count the total number pixels in histogram

    cnt = 0

    for i in range(0, len(h)):

        if h[i]>0:

           cnt += h[i]

    return cnt


def weight(h,s, e): # This is used to get summation of weights from s to e
in probability histogram of image

    w = float(0)

    # print(s,e)

    for i in range(s, e):

        w += h[i]

    return w



def mean(h,s, e): # This is used to get mean from s to e in probability
histogram of image

    m = float(0)
    w = weight(h,s, e)
    for i in range(s, e):
        m += h[i] * i

    return m/w

def variance(h,s, e):# This is used to get variance from s to e in
probability histogram of image
    v = float(0)
    m = mean(h,s, e)
    w = weight(h,s, e)
    for i in range(s, e):
        v += ((i - m) **2) * h[i]
    v /= w
    return v

def threshold(h):# This is find within-class and between class variances
at all intensity levels in probability histogram
    threshold_values = {}
    vwc_values=np.zeros((256,))
    vbc_values=np.zeros((256,))
    for i in range(1, len(h)):
        vb = variance(h,0, i)
        wb = weight(h,0, i)
        mb = mean(h,0, i)
        vf = variance(h,i, len(h))
        wf = weight(h,i, len(h))
        mf = mean(h,i, len(h))
        vwc = wb * (vb) + wf * (vf)

        vbc = wb * wf * (mb - mf)**2



        if not math.isnan(vwc):

            threshold_values[i] = vwc

            vwc_values[i]=vwc


        if not math.isnan(vbc):

            vbc_values[i]=vbc


    x = np.arange(0,256)

    plt.figure(figsize=(10,5))

    plt.plot(x, vwc_values,label="Within Class Variance")

    plt.plot(x, vbc_values,label="Between Class Variance")

    plt.xlabel("Variance")

    plt.ylabel("Intensity")

    plt.legend()

    plt.show()

    return threshold_values


def get_optimal_threshold(threshold_values):# This is used to find a
threshold that minimizes within class variance.

    min_V2w = min(threshold_values.values())

    optimal_threshold = [k for k, v in threshold_values.items() if v ==
min_V2w]

    print('optimal threshold', optimal_threshold[0])

    return optimal_threshold[0]
```

```
def Otsu_Threshold(img): # This is driver code to call all function and
implement final Algorithm given image img
    img = np.asarray(img)
    h = Hist(img)
    threshold_values =threshold(h)
    op_thres = get_optimal_threshold(threshold_values)
    res = regenerate_img(img, op_thres)
    return res
```

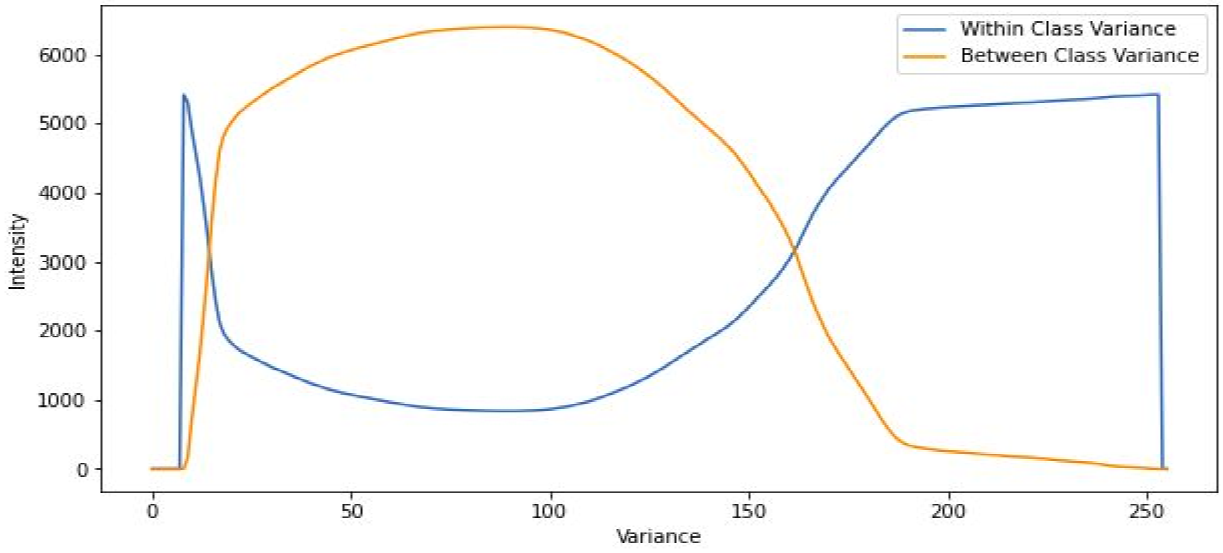
Result and Analysis:

I have tested the algorithm in the following image, cameraman :





Above figure show probability histogram of Cameraman Image

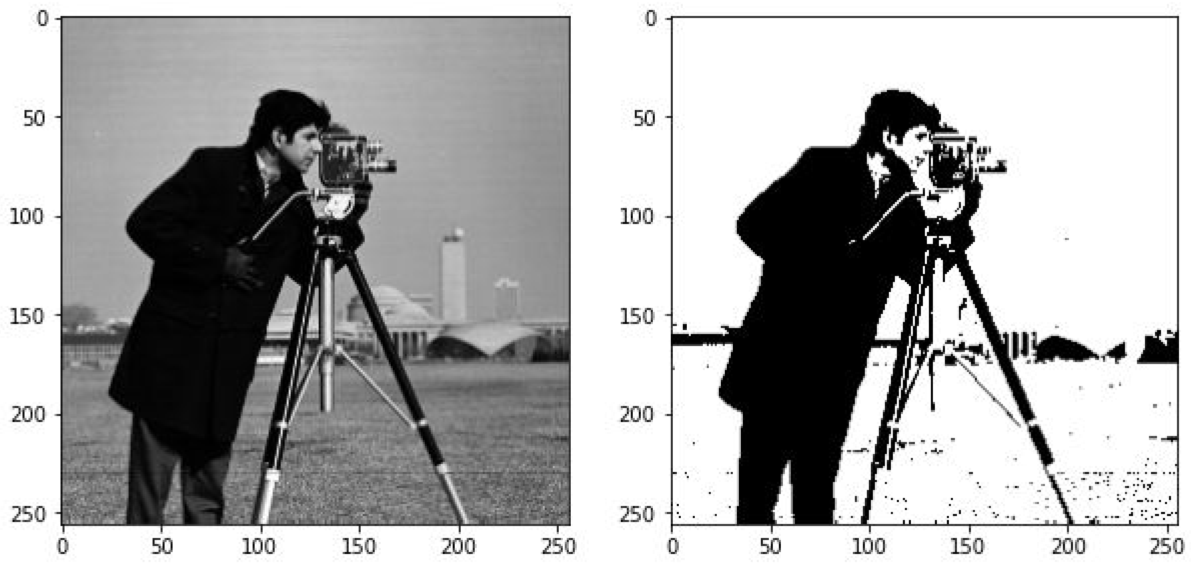


Above Image shows values of Different variances in cameraman image

(Note that 0 values and Undefines(Nan))

I got Maximum Between class variance at 89th Intensity Level.

Thresholding (i.e making pixel less 89 intensity white and greater black) using that value gives following binarized image.

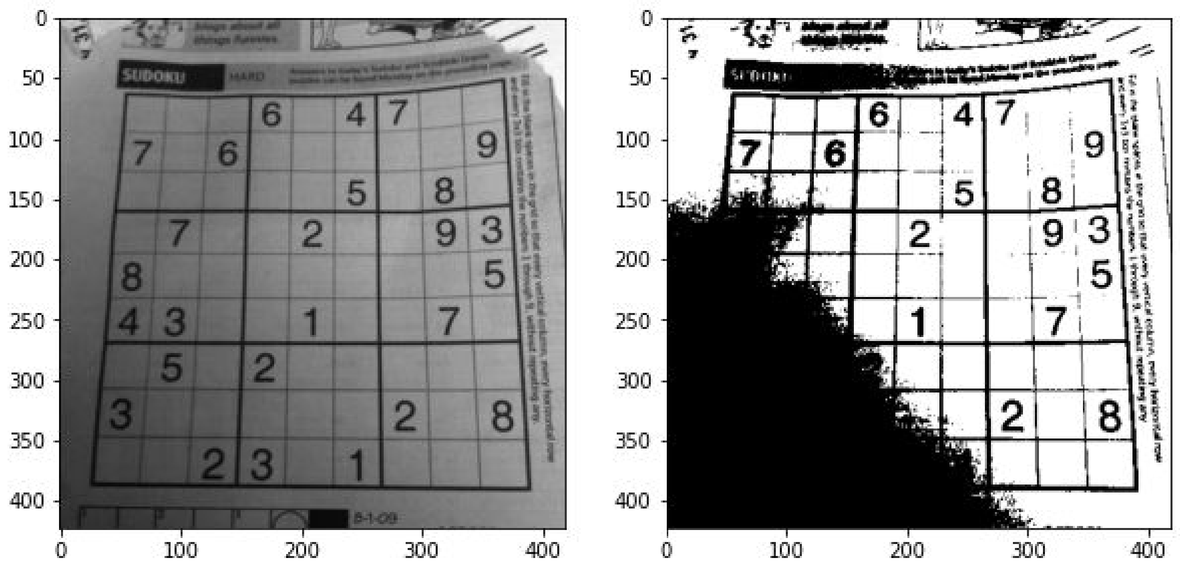


Above image shows Global Binarization of Cameraman Image.

The drawback of Global Thresholding:

The problem with global thresholding is that changes in illumination across the scene may cause some parts to be brighter (in the light) and some parts darker (in shadow) in ways that have nothing to do with the objects in the image because The correct maximum is not necessarily the global one.

For example global Otsu fails to capture sudoku grid in following image:



Above image shows Global Binarization of Sudoku Image. We can observe black patches , some part background i.e page has also merged in foreground i.e number is puzzle which is not a good segmentation of number from pages. This is drawback in global thresholding we address this problem in next section

Using Local Threshold to improve segmentation:

Instead of having a single global threshold, we allow the threshold itself to smoothly vary across the image. We choose a block and divide images blocks and use otsu method on individual blocks to improve binarization. In this way small regions will have their own thresholding set optimally which avoid them being white or black completely.

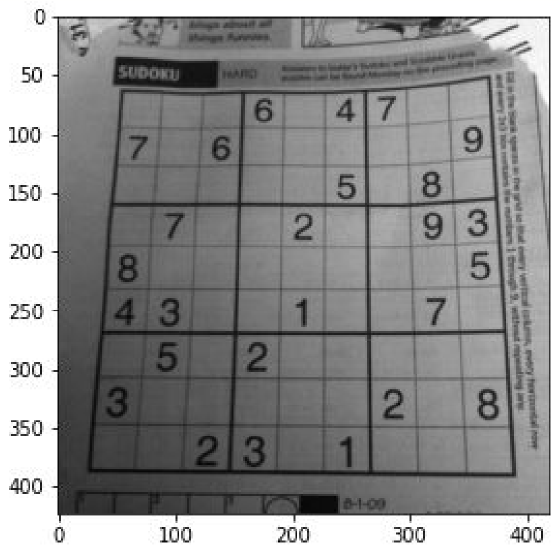
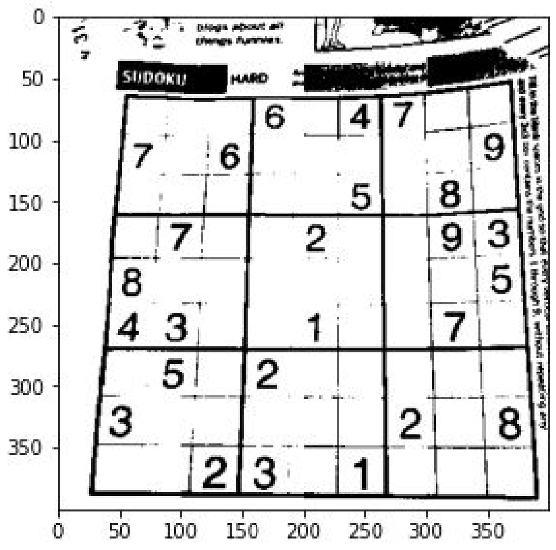
Implementation:

Below is Implementation of Local Thresholding using Otsu's method we assume the function implemented previously.

```
#Local Thresholding
def Local_Threshold(img,block_size): #Local Thresholding will take image
and block size and apply Otsu’s method on each block
   img = np.asarray(img)
   print(img.shape)
   row, col = img.shape
   y = np.zeros((row, col))
   windowsize_r = block_size
   windowsize_c = block_size
   # Crop out the window and calculate the local threshold
   for r in range(0,img.shape[0] - windowsize_r, windowsize_r):
       for c in range(0,img.shape[1] - windowsize_c, windowsize_c):
           window = img[r:r+windowsize_r,c:c+windowsize_c]
           y[r:r+windowsize_r,c:c+windowsize_c]=Otsu_Threshold(window)
   x_lim=(col//block_size)*block_size
   y_lim=(row//block_size)*block_size
```

```
  print(x_lim,y_lim)

  return y[0:x_lim,0:y_lim]
```

Results and Analysis:

Above is local thresholding applied in each block of size 100 in the image.

We can observe that dark regions of image produced by global thresholding are not in local thresholding and we get better segmentation of sudoku numbers from paper background.

Conclusion:

Here I have Achieved following milestones:

I. Implemented Otsu's method in Python with visualisation and analysis of results.

II. Addressed drawback in global thresholding and implement a local threshold using Otsu's method to improve binarization.

Document image binarization is the most important step in pre-processing of scanned documents to save all or maximum subcomponents such us text, background and image

And the Otsu method is one the most widely used techniques for image binarization.

Using Local thresholding further improves the Otsu method but it Also comes with extra computation and applying algorithms to each block. We can choose global thresholding or local one depending on our applications.

Complete Implementation and Images used can be found in following link:

https://github.com/Prateek1337/Otsu_Thresholding

Reference

1. https://engineering.purdue.edu/kak/computervision/ECE661.08/OTSU_paper.pdf
2. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MORSE/threshold.pdf