Introduction to Pattern Recognition WS 2018/19
Franziska Schirrmacher, franziska.schirrmacher@fau.de

Programming Task 3: Otsu's Thresholding and Canny Edge Detector (10 Points)

Deadline: 09. December 2018

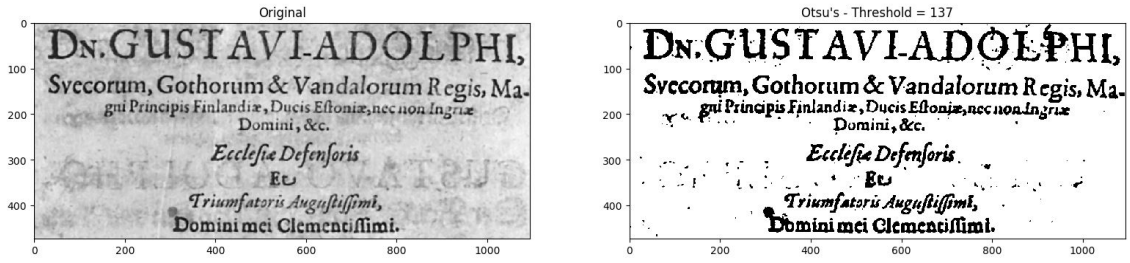
1 Otsu's Thresholding (3 P)

Many thresholding algorithms exist for image binarization where the goal of this process is to split an image into two parts. For example in document processing, the document has to be split into page and writings. One widely used approach is Otsu's thresholding, which was also discussed in the lecture. In this task, you have to implement Otsu's thresholding according to the lecture slides. Use the code skeleton provided. You are only allowed to use the predefined imported modules (in this case - numpy only). The function should work by running the given *main.py* module.

The steps of Otsu's are as follows:

1. Create Histogram from image with 256 bins
2. For every bin calculate:
   (a) Calculate p_0, p_1
   (b) Calculate μ_0, μ_1
   (c) Calculate between class variance $\theta = p_0\, p_1\, (\mu_1 - \mu_0)^2$
3. The bin with the highest between class variance is the threshold
4. Binarize the image (0,255)

Output Example



2 Canny Edge Detection (7 P)

In this task, we want to implement the Canny Edge Detector on our own. The approach consists of several steps that we want to implement subsequently. Use *CannyEdgeDetector.py* and complete the missing functions. Do not change the function *canny*(*img*) but write your code such that it works by calling this function via the *main.py* module. Of course, you are allowed to play around with the parameters within the *canny*(*img*) function. Again, it is not allowed to import other modules!

2.1 Denoising

The first step is to denoise the image. Implement it in $gaussFilter(image, ksize, sigma)$ and return the filtered image.

We want to use a Gaussian filter with a variable kernel size $ksize$. The equation for the kernel matrix is

$$G(x,y) = \frac{1}{2\pi\sigma^2} exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{1}$$

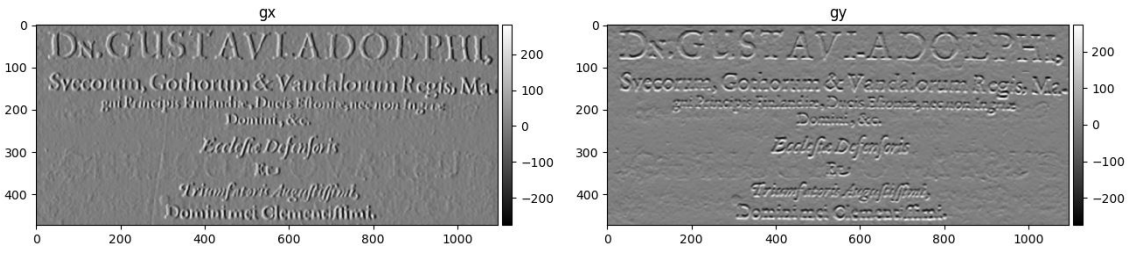
where (x, y) is the current pixel position and σ is the control parameter. x^2+y^2 calculates the distance of the current pixel from the center pixel in the matrix. **Assure that the matrix sums up to 1.** Convolve the image with the kernel.
Example:
Consider a kernel size of $k = 3$ and $\sigma = 2$. The calculation would be as follows:

$$kernel = \begin{pmatrix} \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{1^2+1^2}{2\cdot 2^2}\right) & \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{0^2+1^2}{2\cdot 2^2}\right) & \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{1^2+1^2}{2\cdot 2^2}\right) \\ \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{1^2+0^2}{2\cdot 2^2}\right) & \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{0^2+0^2}{2\cdot 2^2}\right) & \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{1^2+0^2}{2\cdot 2^2}\right) \\ \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{1^2+1^2}{2\cdot 2^2}\right) & \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{0^2+1^2}{2\cdot 2^2}\right) & \frac{1}{2\cdot\pi\cdot 2^2} exp\left(-\frac{1^2+1^2}{2\cdot 2^2}\right) \end{pmatrix} \tag{2}$$

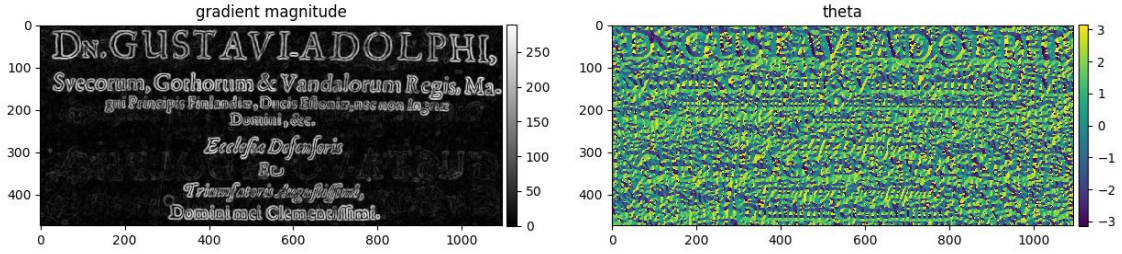
You still have to normalize the matrix such that all elements sum up to 1 and convolve the image with the kernel matrix by using *scipy.ndimage.convolve*(*image*, *kernel*).

2.2 Calculate the Gradient Magnitude and Direction

The next step is to calculate the gradient magnitude and direction. This is done by convolving the image with a sobel filter in x and y direction to receive g_x and g_y which are the gradient images in x and y direction. Implement the sobel filter according to the lecture slides (*scipy.ndimage.convolve*) in the function *sobel*(*img*) and return (g_x, g_y).



Next, calculate the gradient magnitude $g = \sqrt{g_x^2 + g_y^2}$ and the direction of the gradient $theta = arctan2(g_y, g_x)$ in $gradientAndDirection(gx, gy)$.



2.3 Maximum Suppression

In $maxSuppress(g, theta)$ you have to implement the maximum suppression. This should be done by processing the following steps for every pixel (x, y) of g:

1. Convert angle stored in $theta(x, y)$ to degree's and convert it to be within the interval of $[0, 180]$ (Example: $theta = 179 \rightarrow$ fits, $theta = 359 \rightarrow$ convert to 179, $theta = 716$ - convert to 179).

2. Afterwards we want to round the the angle to one of the four angles $0, 45, 90, 135$ by finding it's nearest neighbor (Example $theta = 22 \rightarrow$ round to 0, $theta = 179 \rightarrow$ round to 0). Angle meanings: $theta = 0$ - gradient horizontal. $theta = 45$ - gradient diagonal (down left to up right). $theta = 90$ - gradient vertical. $theta = 135$ - gradient diagonal (up left to down right).

3. Lastly, we want to get the maximal values only. Therefore, we have to store only the local maxima by searching for them according to the gradient direction. (Example: $theta = 0 \rightarrow img(x, y) > img(x+1, y) and img(x, y) > img(x-1, y) \rightarrow localmaxima! \rightarrow store!$ – Example 2: $theta = 135 \rightarrow img(x, y) > img(x+1, y+1) and img(x, y) > img(x-1, y-1) \rightarrow localmaxima! \rightarrow store!$).

Output:

2.4 Hysteris Thresholding

The last step is to implement an hysteris based thresholding. Use $hysteris(img_{in}, t_{low}, t_{high})$ for your implementation. t_{low} is the lower and t_{high} the upper threshold value. First, classify each pixel of the maximum suppressed image by calculating

$$threshimg(x,y) = \begin{cases} 0 & \text{,if } maxsup(x,y) \leq t_{low} \\ 1 & \text{,if } maxsup(x,y) > t_{low} \text{ AND } maxsup(x,y) \leq t_{high} \\ 2 & \text{,if } maxsup(x,y) > t_{high} \end{cases} \tag{3}$$

Afterwards, walk through the classified image, search for pixels that are greater than t_{high} and set them to 255. Also set their neighboring pixels (except border pixels, every pixel has eight neighbors) that are greater than t_{low} to 255. This is the final result that should be returned when using the predefined parameters.

Output: