Umbralización en OpenCV con Python

Umbralización Simple

Aquí, el objetivo es sencillo. Si el valor del pixel es mayor al valor del umbral, se le asigna un valor (puede ser blanco), de otro modo se le asigna otro valor (puede ser negro). La función utilizada es **cv2.threshold**. El primer argumento es la imagen fuente, que **debería encontrarse en escala de grises**. El segundo argumento es el valor del umbral que se usa para calificar los valores de pixeles. El tercer argumento es el **maxVal** el cual representa el valor dado si el valor del pixel es mayor que (a veces menor que) el valor del umbral. *OpenCV* provee diferentes estilos de umbralización y se decide por medio del cuarto parámetro de la función. Los distintos tipos son:

- THRESH_BINARY
- THRESH_BINARY_INV
- THRESH_TRUNC
- THRESH_TOZERO
- THRESH_TOZERO_INV

La documentación explica claramente para qué funciona cada uno. Por favor revisa la documentación para mas info.

Se obtienen dos salidas. La primera es un **retval** el cual explicaremos luego. La segunda es nuestra **imagen umbralizada**.

Código:

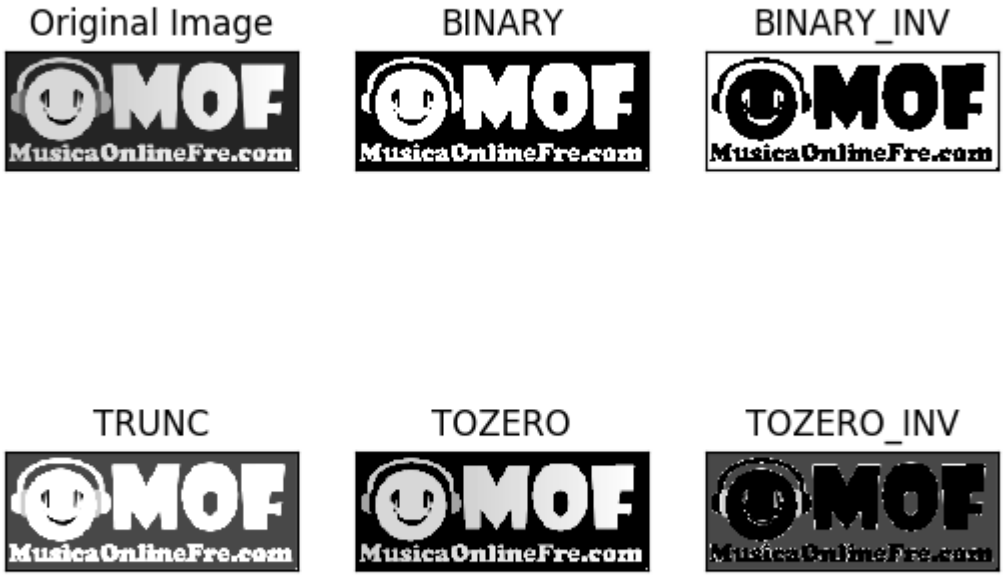
```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('musica.png',0)
ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH_BINARY_INV)
ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO)
ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)

titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
miArray = np.arange(6)
for i in miArray:
  plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
  plt.title(titles[i])
  plt.xticks([]),plt.yticks([])

plt.show()

```

? Nota

Para trazar imágenes múltiples, usamos la función plt.subplot(). Por favor revisa los documentos de Matplotlib para más detalles.

Los resultados obtenidos aquí con el logo de una web de musica

(musicaonlinefre.com)



Umbralización Adaptativa

En la sección previa, usamos un valor global como valor umbral. Pero puede no ser bueno en todos los casos donde las imágenes difieren en cuanto a condiciones de luz en distintas áreas. En ese caso, utilizamos la umbralización adaptativa. En esta, el algoritmo calcula el umbral para una pequeña región de la imagen. Así que obtenemos diferentes umbrales para distintas regiones de la misma imagen. Y nos da mejores resultados para imágenes con iluminación variante.

Posee tres parámetros “especiales” de entrada y sólo un argumento de salida.

Metodo Adaptativo – Decide cómo el valor de umbralización es calculado.

- ADAPTIVE_THRESH_MEAN_C : el valor umbral es equivalente al valor del área vecina.
- ADAPTIVE_THRESH_GAUSSIAN_C : en este caso el valor umbral es la suma de los pesos de los valores vecinos donde dichos valores correspondían a pesos de una ventana gaussiana.

Block Size – Decide el tamaño del área vecina.

C – Es sólo una constante que es substraída del cálculo del medio o el peso del medio calculado.

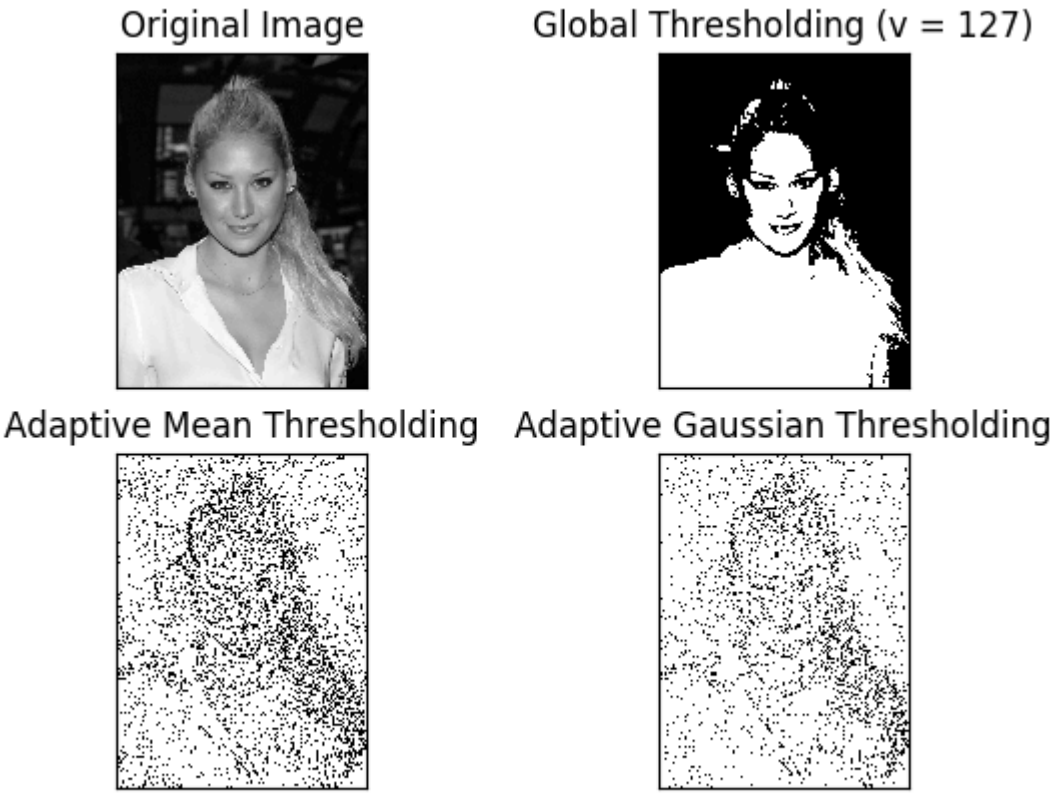
El fragmento de código expresado abajo compara la umbralización global con la adaptativa para una imagen de iluminación variante:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('anna-min.jpg',0)
img = cv2.medianBlur(img,5)

ret,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C,\
 cv2.THRESH_BINARY,11,2)
th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\
 cv2.THRESH_BINARY,11,2)

titles = ['Original Image', 'Global Thresholding (v = 127)',
 'Adaptive Mean Thresholding', 'Adaptive Gaussian Thresholding']
images = [img, th1, th2, th3]
miArray = np.arange(4)
for i in miArray:
 plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
 plt.title(titles[i])
```

```
19      plt.xticks([]),plt.yticks([])
20     plt.show()
21
22
```

Imagen original a color

Resultado



La Binarización de Otsu

En la primera sección, te comenté que había un segundo parámetro denominado **retVal**. Su uso llega cuando usamos la **Binarización de Otsu**. Así que, ¿qué es? En la umbralización global, utilizamos un valor arbitrario como umbral, ¿correcto? Así, ¿cómo podemos saber si el valor que hemos escogido es bueno o no? La respuesta es, mediante el método de ensayo y error. Pero considera una **imagen bimodal** (en pocas palabras, una imagen bimodal es una imagen cuyo histograma posee dos picos). Para esa imagen, podemos tomar un valor aproximado entre esos dos picos como el valor umbral, ¿correcto?

Eso es lo que hace la **binarización de Otsu**. En pocas palabras, se calcula de forma automática un valor de umbral desde el histograma de la imagen bimodal. (Para imágenes que no son bimodales, la binarización no será precisa).

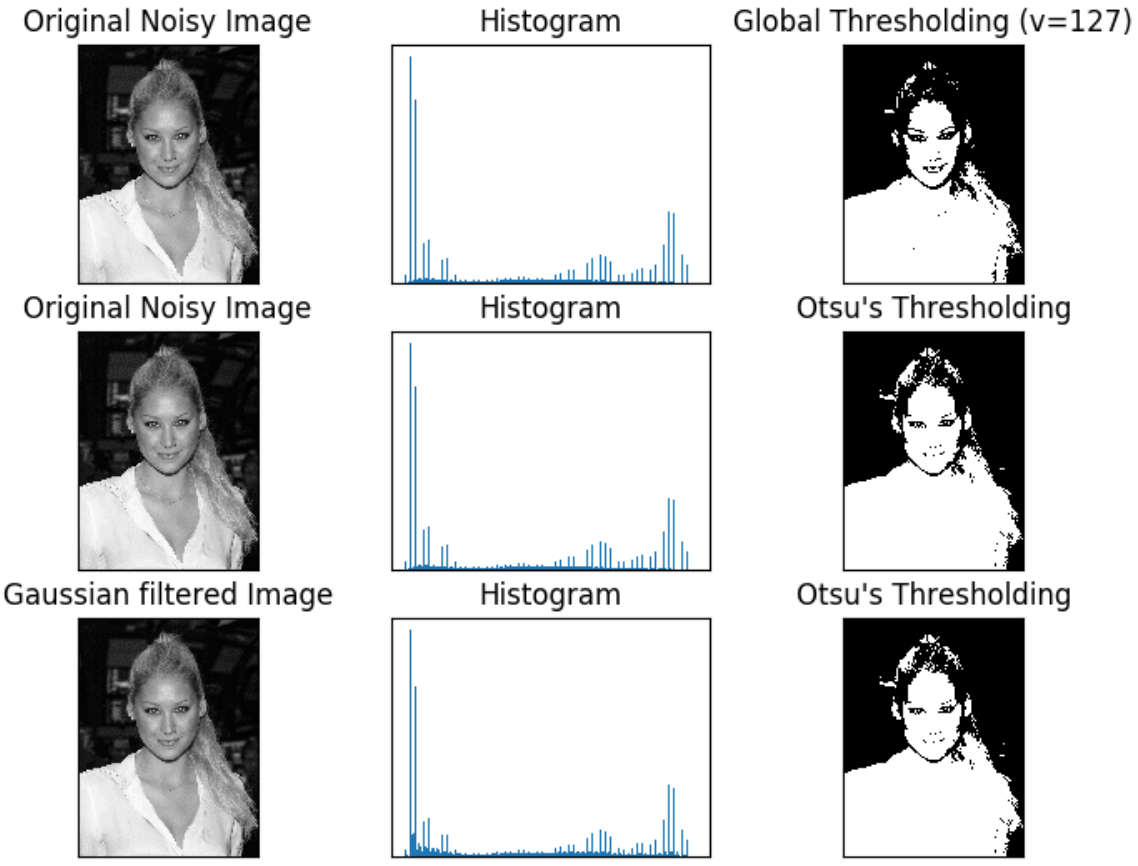
Para esto, usamos nuestra función cv2.threshold(), pero con un indicador adicional, cv2.THRESH_OTSU. **Para el valor umbral, sólo usamos cero.** Luego el algoritmo encuentra el valor umbral óptimo y lo regresa como la segunda salida, retVal. Si la umbralización de Otsu no se usa, retVal es igual al valor de umbral que usaste.

Chequea el ejemplo a continuación. La imagen de entrada posee mucho ruido. En el primer caso, aplico la umbralización global para un valor de 127. En el segundo caso, aplico la umbralización de Otsu de forma directa. En el tercer caso, filtro la imagen con *kernel* gaussiano 5×5 para remover el ruido, luego aplico la umbralización de Otsu. Observa como el filtro que remueve el ruido mejora los resultados.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('anna-min.jpg',0)

# global thresholding
ret1,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)

# Otsu's thresholding
ret2,th2 = cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# Otsu's thresholding after Gaussian filtering
blur = cv2.GaussianBlur(img,(5,5),0)
ret3,th3 = cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# plot all the images and their histograms
images = [img, 0, th1, img, 0, th2, blur, 0, th3]
titles = ['Original Noisy Image','Histogram','Global Thresholding (v=127)',
'Original Noisy Image','Histogram',"Otsu's Thresholding",
'Gaussian filtered Image','Histogram',"Otsu's Thresholding"]
miArray = np.arange(3)
for i in miArray:
```

```
    plt.subplot(3,3,i*3+1),plt.imshow(images[i*3],'gray')
    plt.title(titles[i*3]), plt.xticks([]), plt.yticks([])
    plt.subplot(3,3,i*3+2),plt.hist(images[i*3].ravel(),256)
    plt.title(titles[i*3+1]), plt.xticks([]), plt.yticks([])
    plt.subplot(3,3,i*3+3),plt.imshow(images[i*3+2],'gray')
    plt.title(titles[i*3+2]), plt.xticks([]), plt.yticks([])
plt.show()
```

Resultado



¿Cómo funciona la binarización de Otsu?

Esta sección demuestra una implementación de *Python* de la **binarización de Otsu** para mostrar cómo funciona realmente. Si no te interesa, puedes obviarla.

Dado que estamos trabajando con imágenes bimodales, el algoritmo de Otsu intenta encontrar un valor de umbral (t) que minimice la **varianza de pesos entre clase**, dada por la relación:

$$\sigma_w^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t)$$

donde:

$$q_1(t) = \sum_{i=1}^{t} P(i) \quad \& \quad q_1(t) = \sum_{i=t+1}^{I} P(i)$$

$$\mu_1(t) = \sum_{i=1}^{t} \frac{iP(i)}{q_1(t)} \quad \& \quad \mu_2(t) = \sum_{i=t+1}^{I} \frac{iP(i)}{q_2(t)}$$

$$\sigma_1^2(t) = \sum_{i=1}^{t} [i - \mu_1(t)]^2 \frac{P(i)}{q_1(t)} \quad \& \quad \sigma_2^2(t) = \sum_{i=t+1}^{I} [i - \mu_1(t)]^2 \frac{P(i)}{q_2(t)}$$

En realidad, encuentra un valor de t que yace entre dos picos tales que la varianza entre ambas clases es mínima. Puede ser simplemente implementada en *Pytho*n de la siguiente forma:

```
1  import numpy as np
2  import cv2
   img = cv2.imread('anna-min.png',0)
```

```
blur = cv2.GaussianBlur(img,(5,5),0)

# find normalized_histogram, and its cumulative distribution function
hist = cv2.calcHist([blur],[0],None,[256],[0,256])
hist_norm = hist.ravel()/hist.max()
Q = hist_norm.cumsum()

bins = np.arange(256)

fn_min = np.inf
thresh = -1
miArray = np.arange(256)

for i in miArray:
  p1,p2 = np.hsplit(hist_norm,[i]) # probabilities
  q1,q2 = Q[i],Q[255]-Q[i] # cum sum of classes
  b1,b2 = np.hsplit(bins,[i]) # weights

  # finding means and variances
  m1,m2 = np.sum(p1*b1)/q1, np.sum(p2*b2)/q2
  v1,v2 = np.sum(((b1-m1)**2)*p1)/q1,np.sum(((b2-m2)**2)*p2)/q2

  # calculates the minimization function
  fn= v1*q1 + v2*q2
  if fn < fn_min:
    fn_min = fn
    thresh = i

# find otsu's threshold value with OpenCV function
ret, otsu = cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
print (thresh)
print (ret)
```

(Algunas de las funciones pueden ser nuevas, pero las cubriremos en los próximos capítulos)

Recursos Adicionales

- Procesado de Imágenes Digitales, Rafael C. Gonzalez

Ejercicios

Existen algunas optimizaciones disponibles para la binarización de Otsu. Puedes buscarlas e implementarlas.

Felicitaciones por continuar aprendiendo con nosotros? el manejo de imágenes es un tema amplio y la umbralización de imágenes un aspecto importante de este. En nuestra siguiente lección del **curso Python de OpenCV** veremos como suavizar una imagen.