Untitled2

June 28, 2022

1 Color Image Segmentation — Image Processing

Whenever we see images, it is composed of various elements and objects. In some circumstance, there might come a time that you would want to extract a certain object from the image. What would you do? Ah, I know. The first thing you will think of is to crop it, right? Well, that would somehow work. But there are irrelevant pixels that would be included, and I know for sure you do not want that. What if I tell you that we can obtain objects of interest using image processing techniques?

Welcome to Image segmentation using Python.

1.1 Image Segmentation

It is the process of dividing an image into its constituent parts or objects. Common techniques include edge detection, boundary detection, thresholding, region based segmentation, among others. For this blog, let us focus on segmenting our images using Color Image Segmentation through the HSV color space.

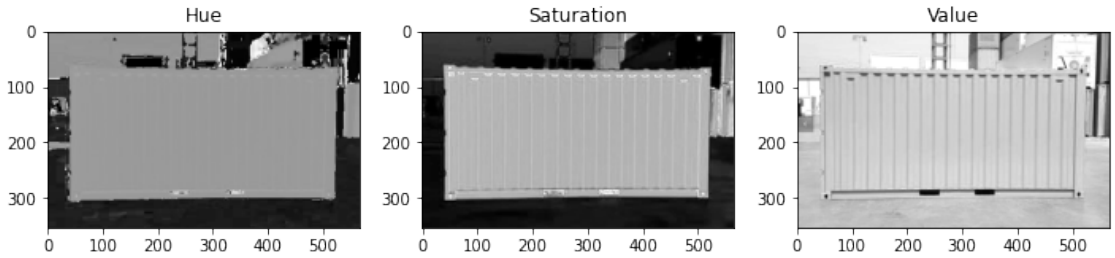
But before we do that, install the following libraries, and follow along.

[1]:
```
# Importing necessary libraries
from skimage import data
from skimage import filters
from skimage import morphology
from skimage.segmentation import clear_border
from skimage.measure import label, regionprops
from skimage.io import imread, imshow
from skimage.color import rgb2hsv, rgb2gray, label2rgb
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import numpy as np




nemo = imread('images/foto2.jpg')
plt.imshow(nemo)
plt.show()
```

Our goal here is to segment container through the HSV color space. But how? As discussed in my previous articles, HSV stands for Hue, Saturation, and Value, and we shall be using information from this color space in segmenting the image later on.

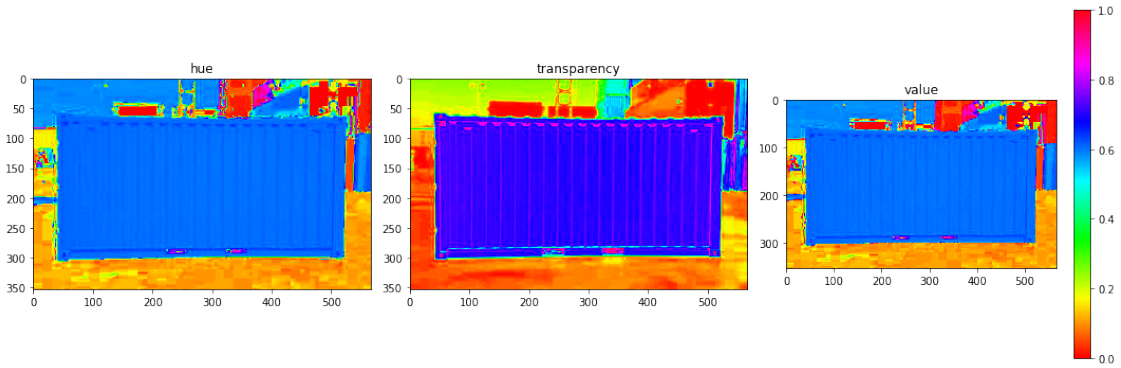
But before the segmentation process, let us first convert the RGB image in HSV form through this code:

[2]:
```
nemo_hsv = rgb2hsv(nemo)

fig, ax = plt.subplots(1, 3, figsize=(12,4))
ax[0].imshow(nemo_hsv[:,:,0], cmap='gray')
ax[0].set_title('Hue')
ax[1].imshow(nemo_hsv[:,:,1], cmap='gray')
ax[1].set_title('Saturation')
ax[2].imshow(nemo_hsv[:,:,2], cmap='gray')
ax[2].set_title('Value');
```



Wonderful! We were successful in transforming the image. However, this is not helpful to us just yet. We need to obtain the intensity values of each HSV channel to help us in segmenting later. To do that, a colorbar is created through implementing this code:

[3]:
```
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

ax[0].imshow(nemo_hsv[:,:,0],cmap='hsv')
ax[0].set_title('hue')

ax[1].imshow(nemo_hsv[:,:,1],cmap='hsv')
ax[1].set_title('transparency')

ax[2].imshow(nemo_hsv[:,:,2],cmap='hsv')
ax[2].set_title('value')

fig.colorbar(imshow(nemo_hsv[:,:,0],cmap='hsv'))
fig.tight_layout()
```



Voila! See the colorbar at the right? We will refer to that in segmenting the bags.

Let us start segmenting

Before implementing the code, let us first set up our thresholds for the masks:

- Lower Mask (refer to the hue channel)
- Upper Mask (refer to the hue channel)
- Saturation Mask (refer to the transparency channel)

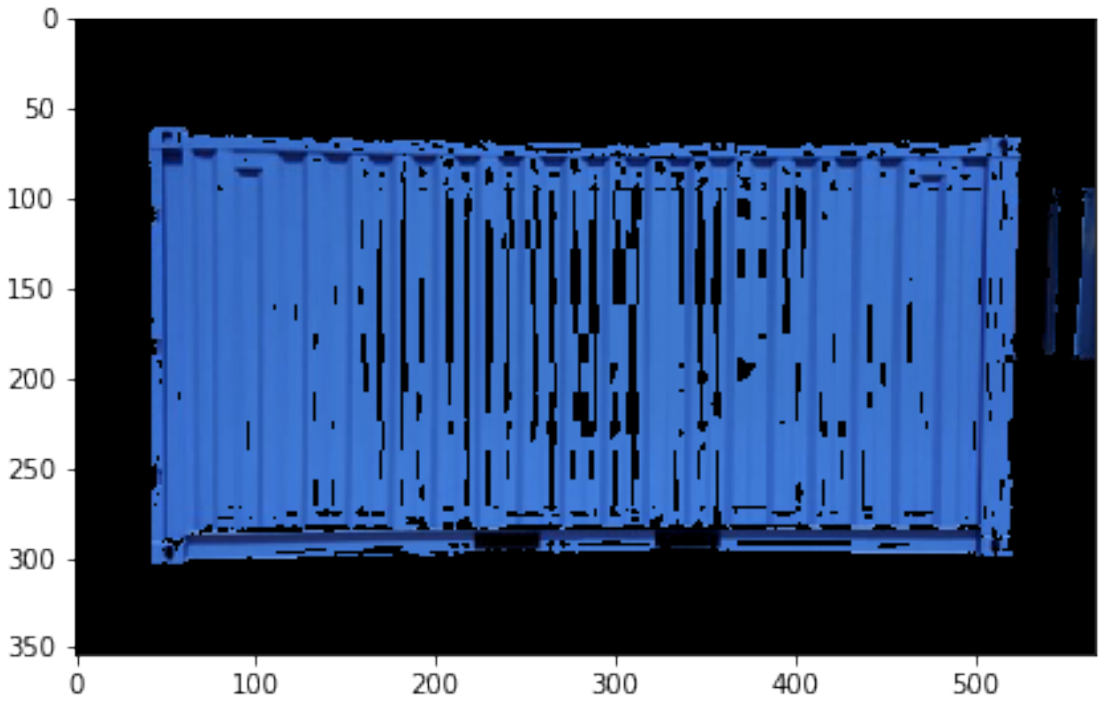
In choosing the thresholds, look at the color bar. For example, if we are segmenting the blue bag. The lower and upper mask values that are appropriate would be 0.6 and 0.7, respectively. So, in other words, it would only get the blue pixel values and neglect the rest. After that, the saturation threshold is decided. This is a bit tricky because you need to consider the colors that are seen in the object. So, play around with the saturation threshold that would return the best segmented image.

Now let us code!

Segmenting the blue bag, we implement this code:

[4]:
```
#refer to hue channel (in the colorbar)
lower_mask = nemo_hsv[:,:,0] > 0.6

#refer to hue channel (in the colorbar)
upper_mask = nemo_hsv[:,:,0] < 0.7

#refer to transparency channel (in the colorbar)
saturation_mask = nemo_hsv[:,:,1] > 0.5

mask = upper_mask*lower_mask*saturation_mask

red = nemo[:,:,0]*mask
green = nemo[:,:,1]*mask
blue = nemo[:,:,2]*mask
nemo_masked = np.dstack((red,green,blue))
imshow(nemo_masked)
```

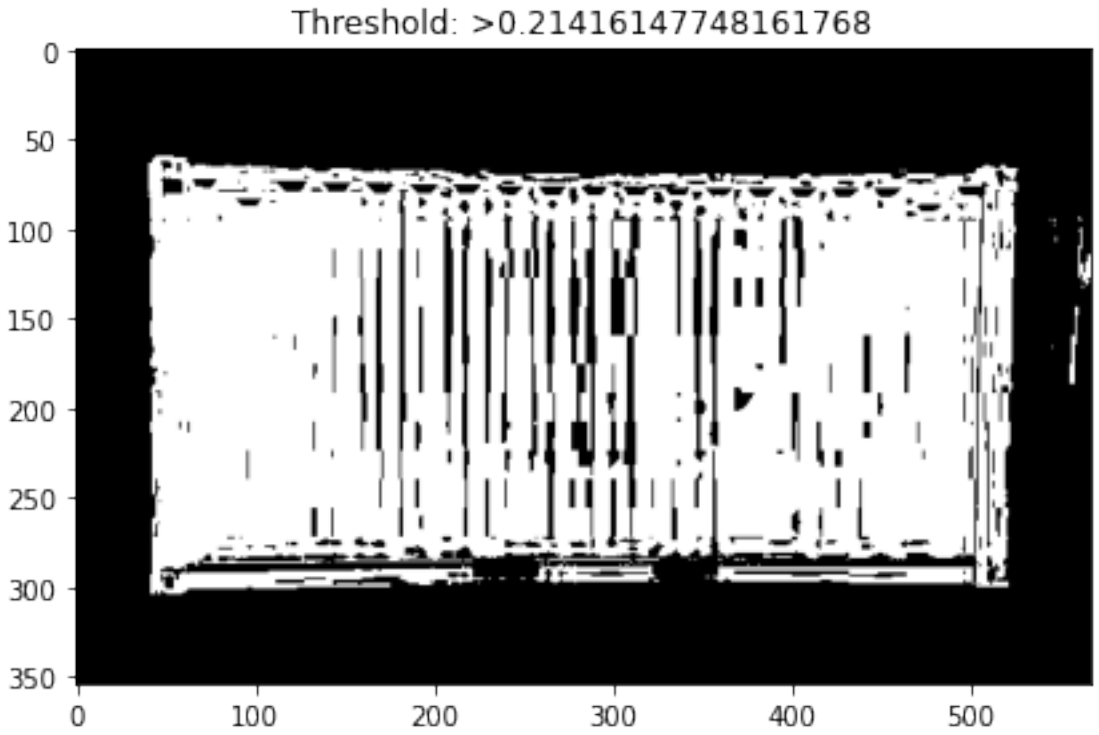
[4]:
```
<matplotlib.image.AxesImage at 0x7fd9c3539550>
```



[5]:
```
# Setting plot size to 15, 15
plt.figure(figsize=(15, 15))
```

```
# To gray
gray_nemo_masked = rgb2gray(nemo_masked)

# Computing Otsu's thresholding value
threshold = filters.threshold_otsu(gray_nemo_masked)

# Computing binarized values using the obtained
# thresold
nemo_bw = (gray_nemo_masked > threshold)*1
plt.subplot(2,2,1)
plt.title("Threshold: >"+str(threshold))
# Displaying the binarized image
plt.imshow(nemo_bw, cmap = "gray")
```

[5]: <matplotlib.image.AxesImage at 0x7fd9c3540d60>



1.2 Connected Components

Instead, we take a look at connected components as the focal point of interest in the analysis. An evident drawback of this approach is that it is heavily reliant on how clean the data is. Hence, the application of tweaking color spaces and the morphological operations would do the trick. See previous article about morphological operations if you're new to that term.

Before we can use the label and region_properties function of skimage of our connected components,

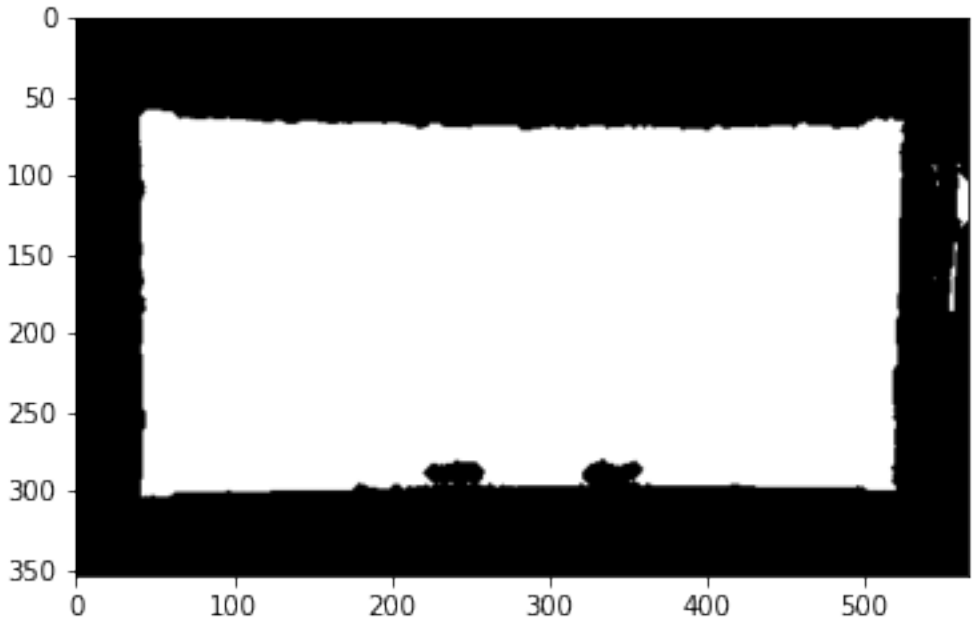
thorough image cleaning must be performed first. Below are user-defined functions to do this:

[6]:
```
def multi_dil(im,num):
  for i in range(num):
      im = morphology.binary_dilation(im)
  return im

def multi_ero(im,num):
  for i in range(num):
      im = morphology.binary_erosion(im)
  return im
```

We use this chain operation to clean the image:

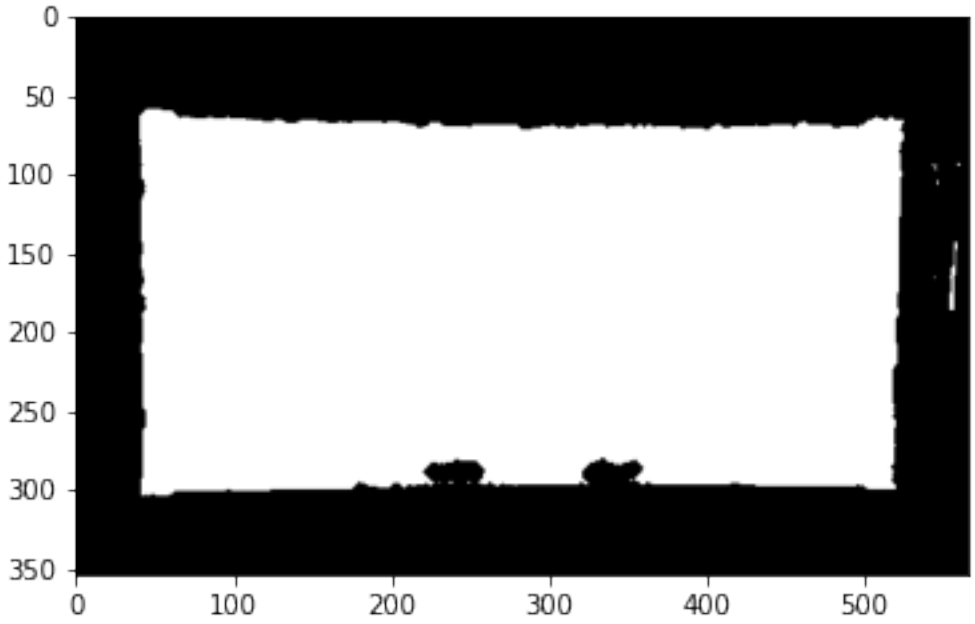
[7]:
```
nemo_cleaned = multi_ero(multi_dil(nemo_bw,5),5)
plt.imshow(nemo_cleaned,cmap='gray')
```

[7]:
```
<matplotlib.image.AxesImage at 0x7fd9c908a0a0>
```



[8]:
```
# remove artifacts connected to image border
nemo_cleaned = clear_border(nemo_cleaned)
plt.imshow(nemo_cleaned,cmap='gray')
```

[8]:
```
<matplotlib.image.AxesImage at 0x7fd9c34fdc70>
```

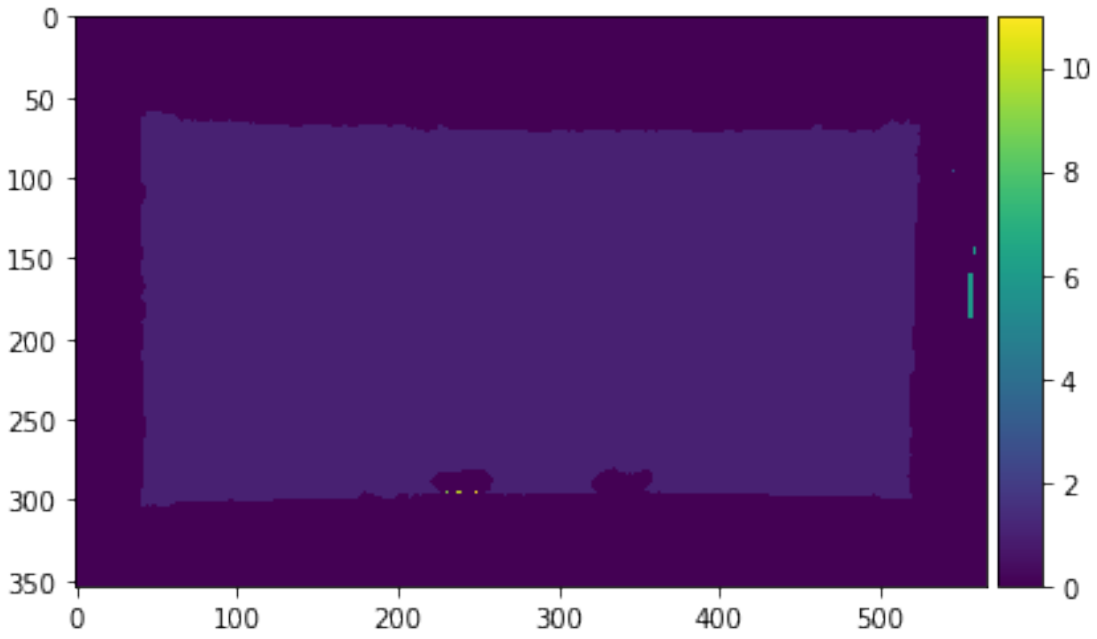


Now that this is relatively clean, let us get the labels and properties of this image!

[9]:
```
label_im = label(nemo_cleaned)
imshow(label_im)
```

```
/home/juan/.virtualenvs/dl4cv/lib/python3.8/site-
packages/skimage/io/_plugins/matplotlib_plugin.py:150: UserWarning: Low image
data range; displaying image with stretched contrast.
  lo, hi, cmap = _get_display_range(image)
```

[9]: `<matplotlib.image.AxesImage at 0x7fd9c37dd7f0>`



There are a lot of features that can be extracted from this labelled image. To mention a few we have:

Now, let's use regionprops and look at the following properties:

1. area
2. perimeter
3. bbox — bounding box dimensions
4. bbox_area — area of bounding box
5. centroid — coordinate of centroid
6. convex_image — convex hull of the blob
7. convex_area — area of the convex hull
8. eccentricity — measure how it fits into an ellipse (0) for circles (how elongated is your object)
9. major_axis_length — length of the major moment of the ellipse fitted
10. minor_axis_length — length of the minor moment of the ellipse fitted

Let us try to take the area of the first object:

[10]:
```
props=regionprops(label_im)

print(len(props))

props[1].area #area (zero) 0th object in the image
props[1].bbox
```

```
11
```

[10]: (94, 560, 96, 562)

[11]:
```
# label image regions
label_image = label(label_im)
# to make the background transparent, pass the value of `bg_label`,
# and leave `bg_color` as `None` and `kind` as `overlay`
image_label_overlay = label2rgb(label_image, image=nemo, bg_label=0)

fig, ax = plt.subplots(figsize=(10, 6))
ax.imshow(image_label_overlay)

for region in regionprops(label_image):
    # take regions with large enough areas
    if region.area >= 100:
        # draw rectangle around segmented object
        minr, minc, maxr, maxc = region.bbox
        rect = mpatches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                                  fill=False, edgecolor='red', linewidth=2)
        ax.add_patch(rect)

ax.set_axis_off()
plt.tight_layout()
plt.show()
```