Image Segmentation using Python's scikit-image module.

An overview of the scikit-image library's image segmentation methods.

Parul Pandey
Feb 15 · 11 min read

source

> Sooner or later all things are numbers, including images.

*People who have seen **The Terminator** would definitely agree that it was the greatest sci-fi movie of that era. In the movie, **James Cameron** introduced an interesting visual effect concept that made it possible for the viewers to get behind the eyes of the cyborg called*

*Terminator. This effect came to be known as the **Terminator Vision** and* in *a way, it segmented humans from the background. It might have sounded totally out of place then, but Image segmentation forms a vital part of many Image processing techniques today.*

. . .

Image Segmentation

We all are pretty aware of the endless possibilities offered by Photoshop or similar graphics editors that take a person from one image and place them into another. However, the first step of doing this is ***identifying where that person is in the source image*** and this is where Image Segmentation comes into play. There are many libraries written for Image Analysis purpose. In this article, we will be discussing in detail about **scikit-image**, a Python-based image processing library.

. . .

> *The entire code can also be accessed from the **Github Repository** associated with this article.*

. . .

Scikit-image

scikit-image.org

Scikit-image is a Python package dedicated to image processing.

Installation

scikit-image can be installed as follows:

```
pip install -U scikit-image(Linux and OSX)
pip install scikit-image(Windows)

# For Conda-based distributions
conda install scikit-image
```

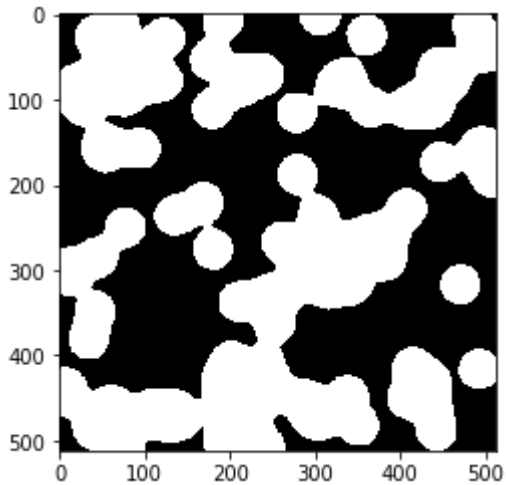
Overview of Images in Python

Before proceeding with the technicalities of Image Segmentation, it is essential to get a little familiar with the scikit image ecosystem and how it handles images.

- **Importing a GrayScale Image from the skimage library**

The skimage data module contains some inbuilt example data sets which are generally stored in jpeg or png format.

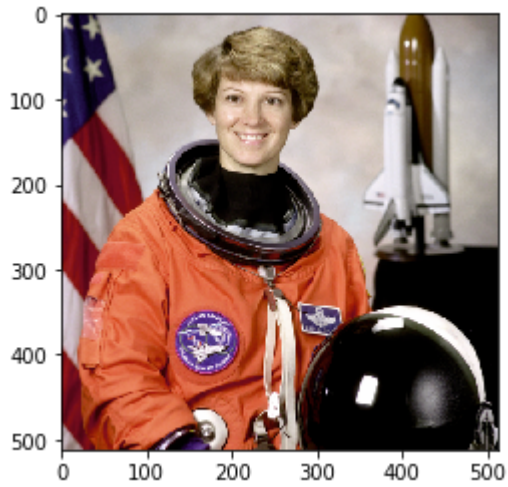
```
from skimage import data
import numpy as np
import matplotlib.pyplot as plt

image = data.binary_blobs()
plt.imshow(image, cmap='gray')
```



- **Importing a Colored Image from the skimage library**

```
from skimage import data
import numpy as np
import matplotlib.pyplot as plt

image = data.astronaut()
```

```
plt.imshow(image)
```



- **Importing an image from an external source**

```
# The I/O module is used for importing the image
from skimage import data
import numpy as np
import matplotlib.pyplot as plt
from skimage import io

image = io.imread('skimage_logo.png')

plt.imshow(image);
```

- **Loading multiple images**

```
images = io.ImageCollection('../images/*.png:../images/*.jpg')

print('Type:', type(images))
images.files
Out[]: Type: <class ‘skimage.io.collection.ImageCollection’>
```

- **Saving images**

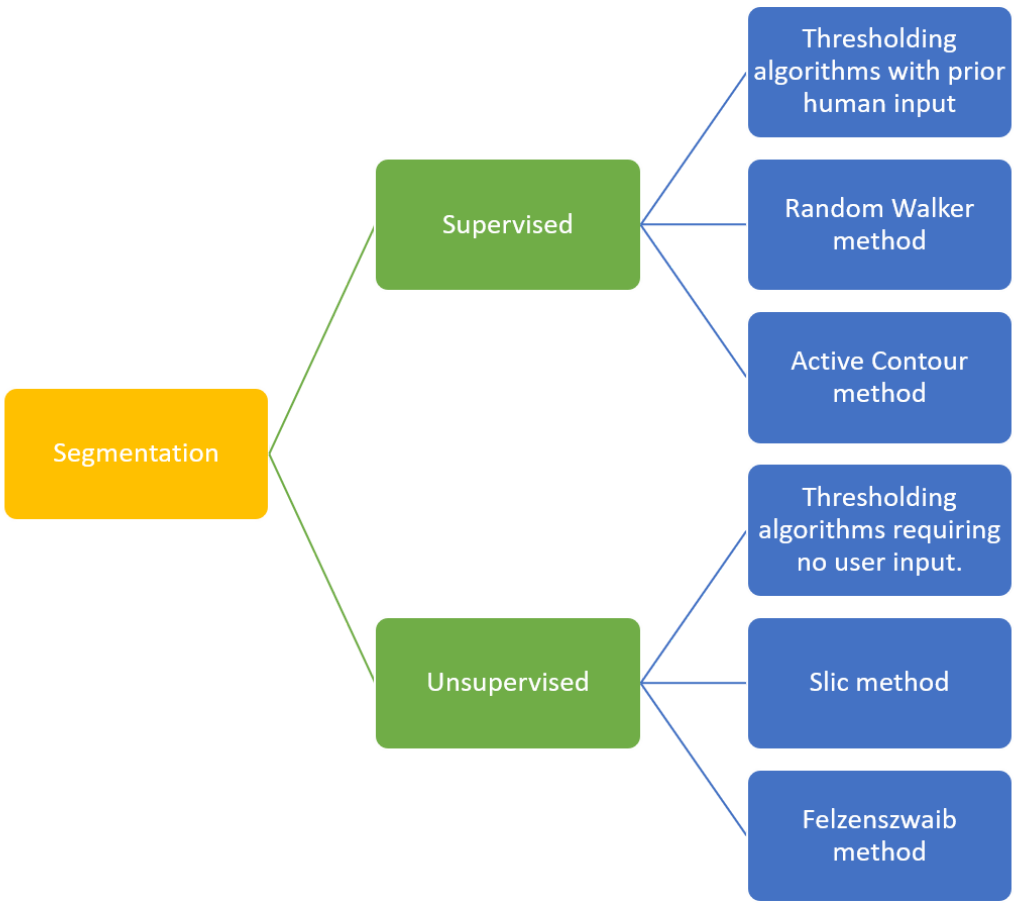
```
#Saving file as ‘logo.png’
io.imsave('logo.png', logo)
```

. . .

Image segmentation

Now that we have an idea about scikit-image, let us get into details of Image Segmentation. **Image Segmentation** is essentially the process of partitioning a digital image into multiple segments to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze.

In this article, we will approach the Segmentation process as a combination of Supervised and Unsupervised algorithms.



Some of the Segmentation Algorithms available in the scikit-image library

Supervised segmentation: Some prior knowledge, possibly from human input, is used to guide the algorithm.

Unsupervised segmentation: No prior knowledge is required. These algorithms attempt to subdivide images into meaningful regions automatically. The user may still be able to tweak certain settings to obtain desired outputs.

Let's begin with the simplest algorithm called **Thresholding.**

Thresholding

It is the simplest way to segment objects from a background by choosing pixels above or below a certain ***threshold.*** This is generally helpful when we intend to segment objects from their background. You can read more about thresholding here.

Let's try this on an image of a textbook which comes preloaded with the scikit-image dataset.

Basic Imports

```
import numpy as np
import matplotlib.pyplot as plt

import skimage.data as data
import skimage.segmentation as seg
import skimage.filters as filters
import skimage.draw as draw
import skimage.color as color
```

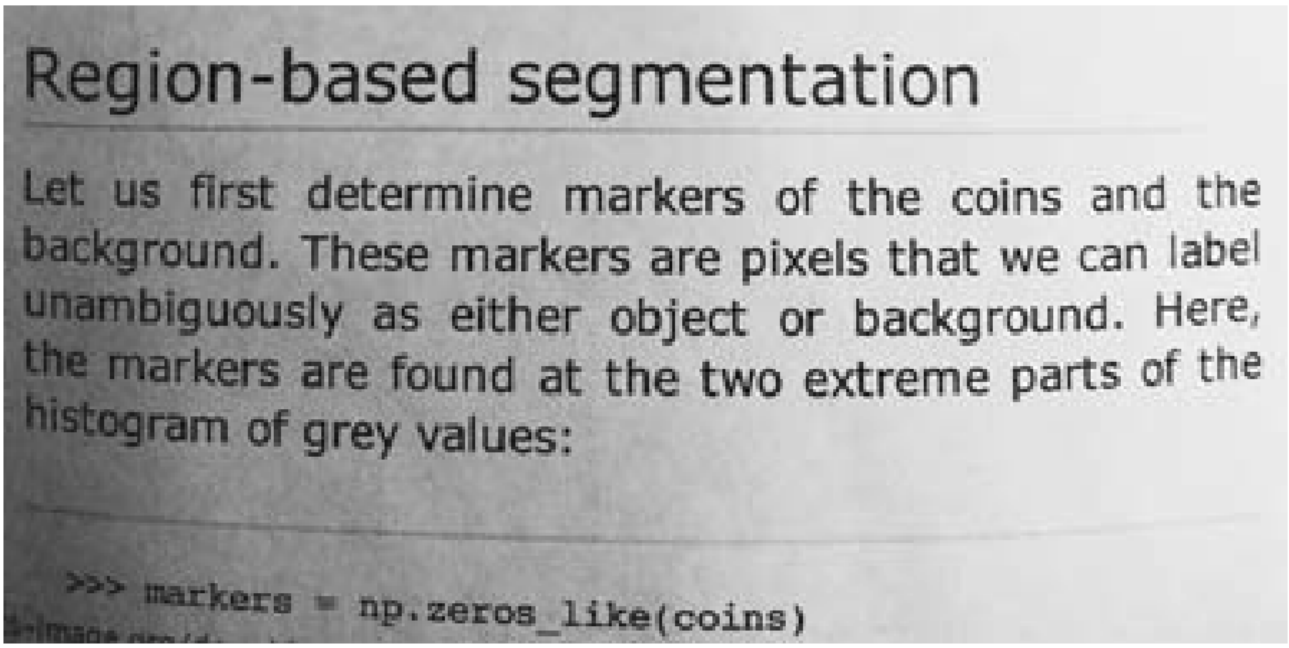
A simple function to plot the images

```
def image_show(image, nrows=1, ncols=1, cmap='gray'):
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=(14,
14))
    ax.imshow(image, cmap='gray')
    ax.axis('off')
    return fig, ax
```

Image

```
text = data.page()
```

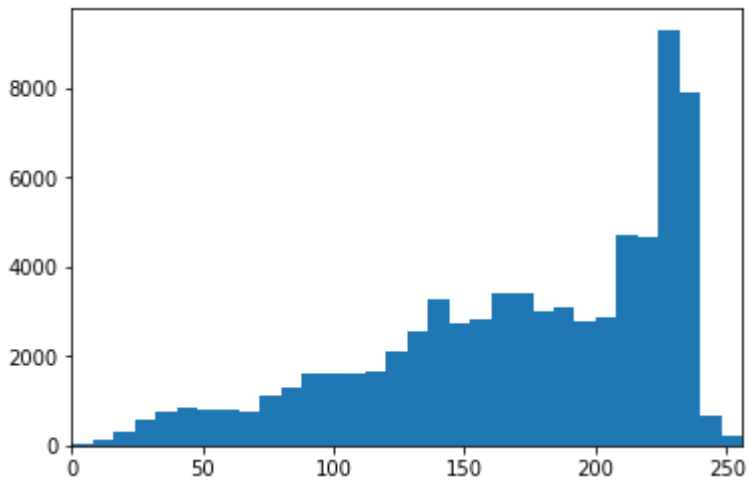
```
image_show(text)
```



This image is a little darker but maybe we can still pick a value that will give us a reasonable segmentation without any advanced algorithms. Now to help us in picking that value, we will use a **Histogram**.

A histogram is a graph showing the number of pixels in an image at different intensity values found in that image. Simply put, a histogram is a graph wherein the x-axis shows all the values that are in the image while the y-axis shows the frequency of those values.

```
fig, ax = plt.subplots(1, 1)
ax.hist(text.ravel(), bins=32, range=[0, 256])
ax.set_xlim(0, 256);
```

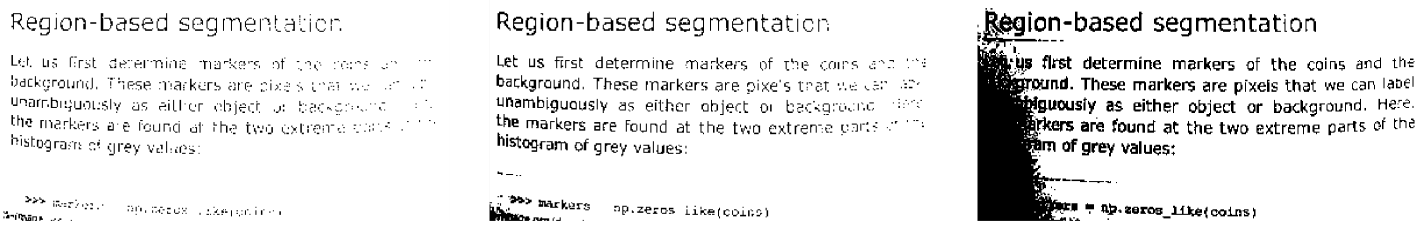


Our example happens to be an 8-bit image so we have a total of 256 possible values on the x-axis. We observe that there is a concentration of pixels that are fairly light(0: black, 255: white). That's most likely our fairly light text background but then the rest of it is kind of smeared out. An ideal segmentation histogram would be bimodal and fairly separated so that we could pick a number right in the middle. Now, let's just try and make a few segmented images based on simple thresholding.

Supervised thresholding

Since we will be choosing the thresholding value ourselves, we call it supervised thresholding.

```
text_segmented = text > (value concluded from histogram i.e
50,70,120 )

image_show(text_segmented);
```



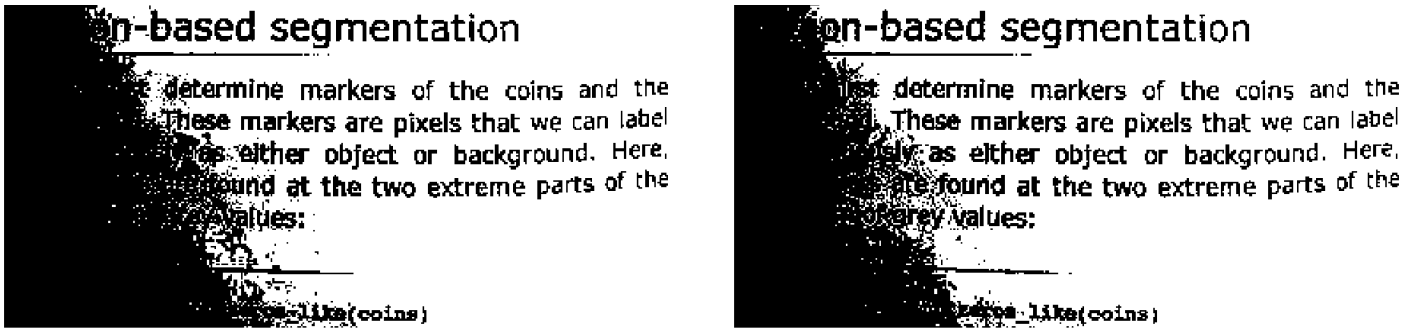
Left: text>50 | Middle : text > 70 | Right : text >120

We didn't get any ideal results since the shadow on the left creates problems. Let's try with unsupervised thresholding now.

Unsupervised thresholding

Scikit-image has a number of automatic thresholding methods, which require no input in choosing an optimal threshold. Some of the methods are : `otsu, li, local.`

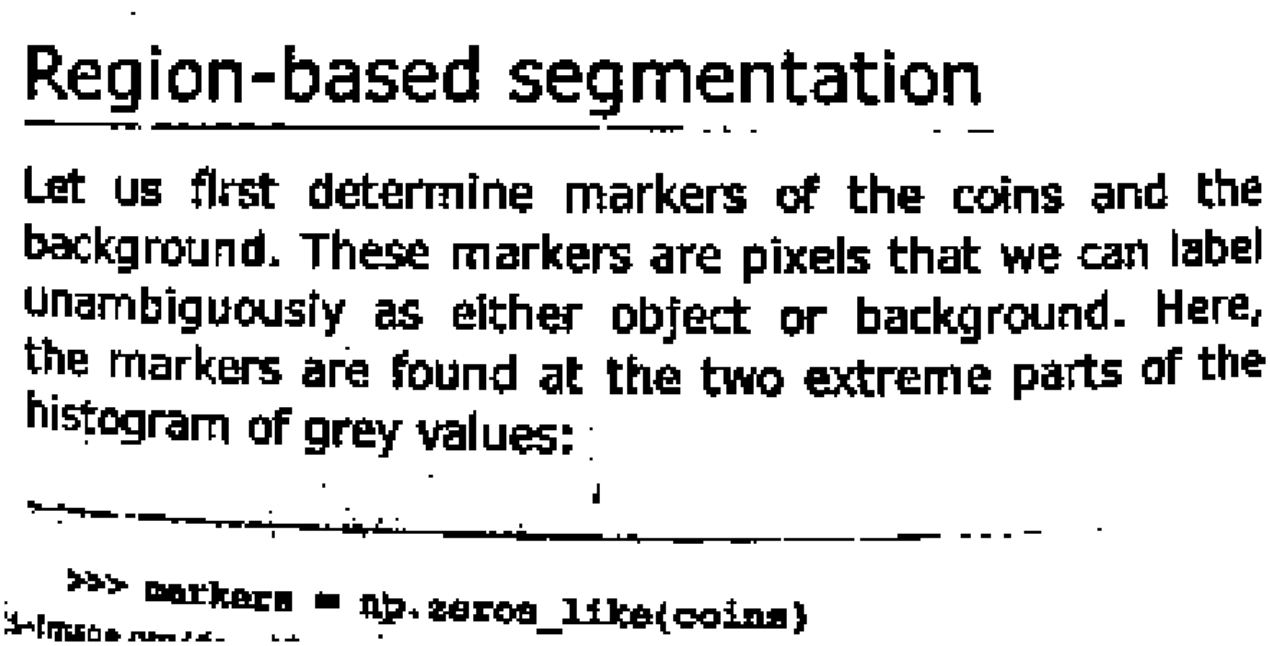
```
text_threshold = filters.threshold_  # Hit tab with the cursor after
the underscore to get all the methods.

image_show(text < text_threshold);
```



Left: otsu || Right: li

In the case of `local`, we also need to specify the `block_size`. `Offset` helps to tune the image for better results.

```
text_threshold = filters.threshold_local(text,block_size=51,
offset=10)
image_show(text > text_threshold);
```



local thresholding

This is pretty good and has got rid of the noisy regions to a large extent.

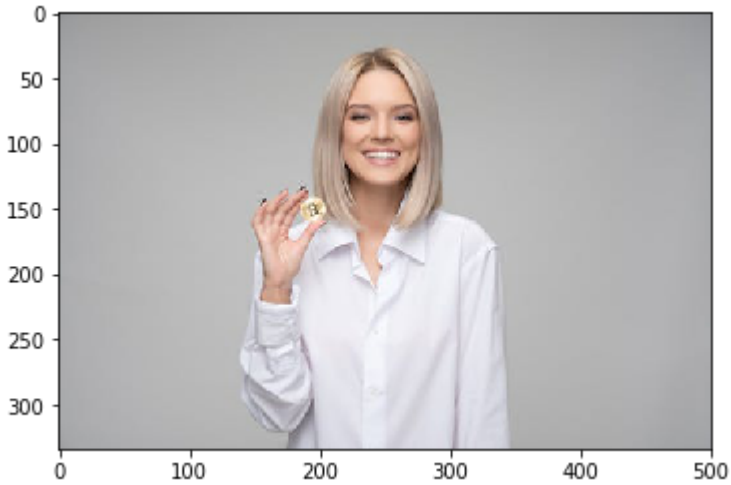
. . .

Supervised segmentation

Thresholding is a very basic segmentation process and will not work properly in a high-contrast image for which we will be needing more advanced tools.

For this section, we will use an example image which is freely available and attempt to segment the head portion using supervised segmentation techniques.

```
# import the image
from skimage import io
image = io.imread('girl.jpg')
plt.imshow(image);
```



source

> Before doing any segmentation on an image, it is a good idea to de-noise it using some filters.

However, in our case, the image is not very noisy, so we will take it as it is. Next step would be to convert the image to grayscale with `rgb2gray`.

```
image_gray = color.rgb2gray(image)
image_show(image_gray);
```

We will use two segmentation methods which work on entirely different principles.

Active contour segmentation

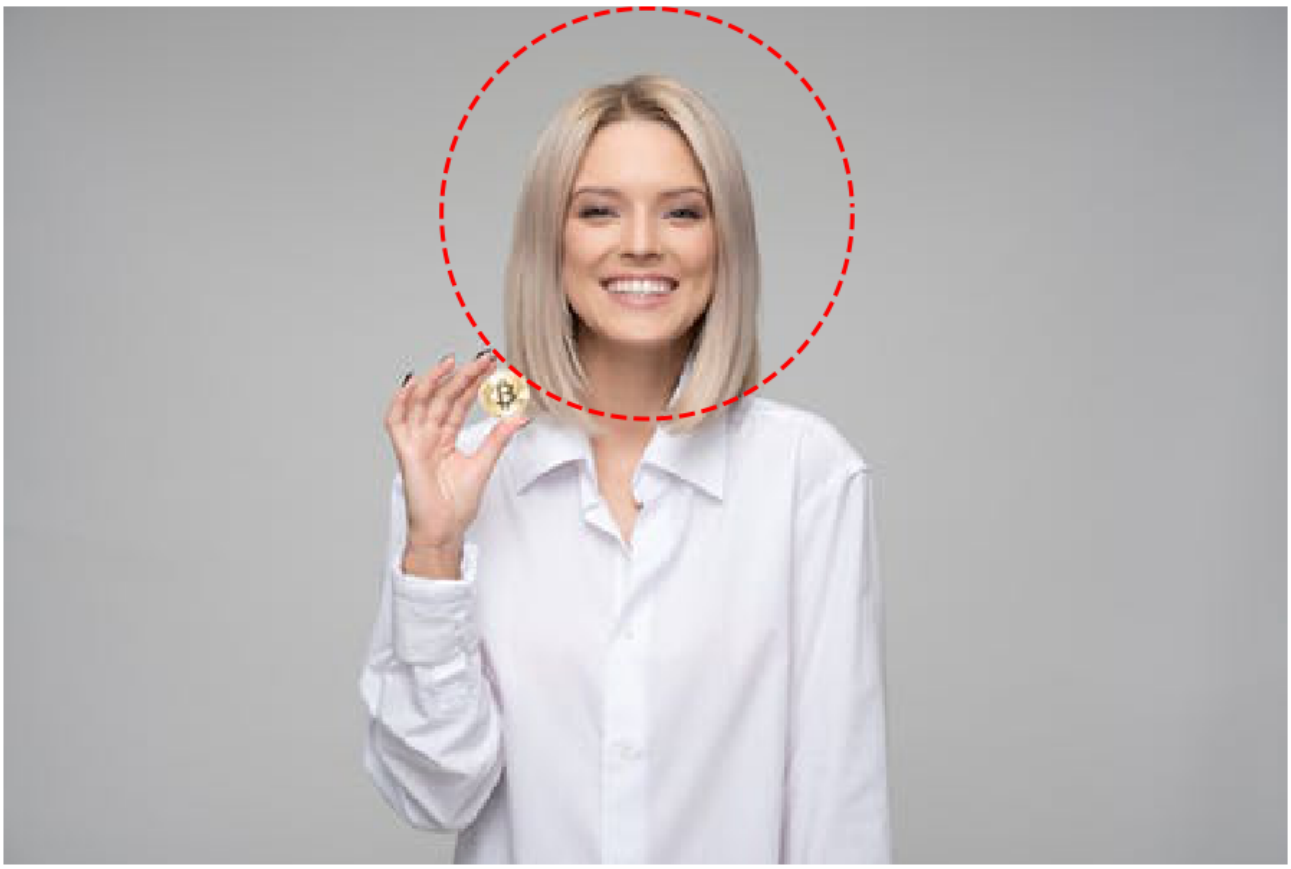
Active Contour segmentation also called as **snakes** and is initialized using a user-defined contour or line, around the area of interest and this contour then slowly contracts and is attracted or repelled from light and edges.

For our example image, let's draw a circle around the person's head to initialize the snake.

```
def circle_points(resolution, center, radius):

    """
    Generate points which define a circle on an image.Centre refers
to the centre of the circle
    """
    radians = np.linspace(0, 2*np.pi, resolution)

    c = center[1] + radius*np.cos(radians)#polar co-ordinates
    r = center[0] + radius*np.sin(radians)

    return np.array([c, r]).T

# Exclude last point because a closed path should not have duplicate
points
points = circle_points(200, [80, 250], 80)[:-1]
```

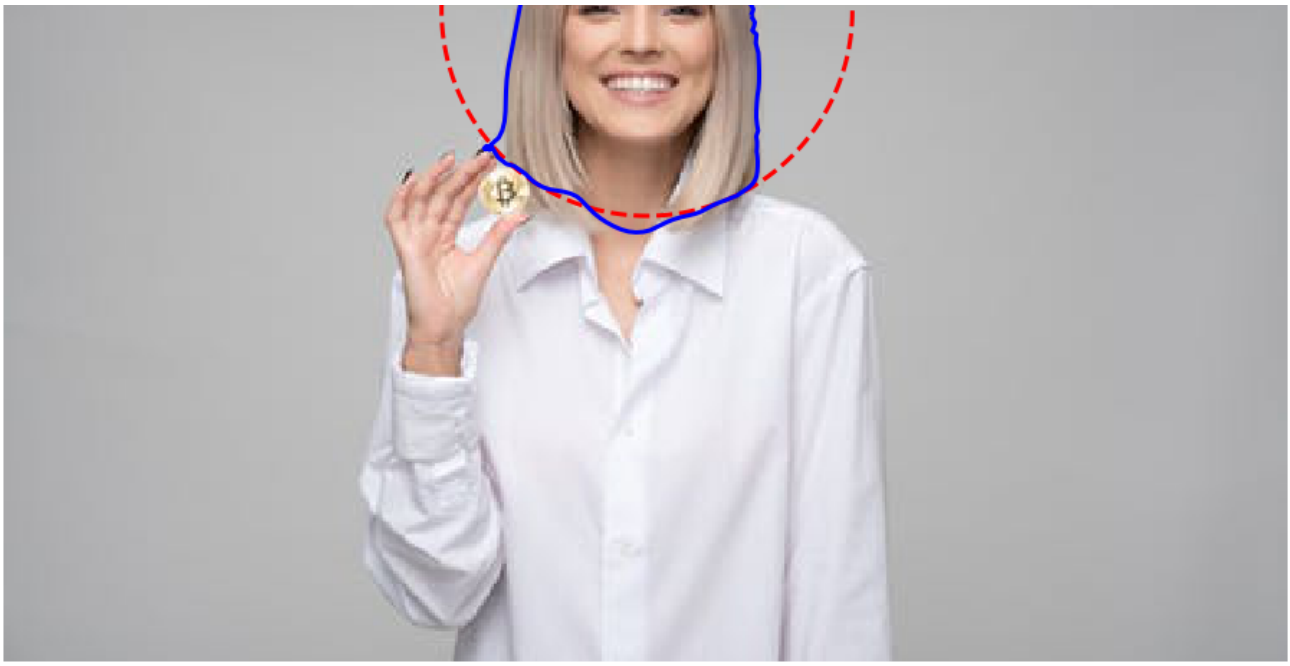
> *The above calculations calculate x and y co-ordinates of the points on the periphery of the circle. Since we have given the resolution to be 200, it will calculate 200 such points.*

```
fig, ax = image_show(image)
ax.plot(points[:, 0], points[:, 1], '--r', lw=3)
```



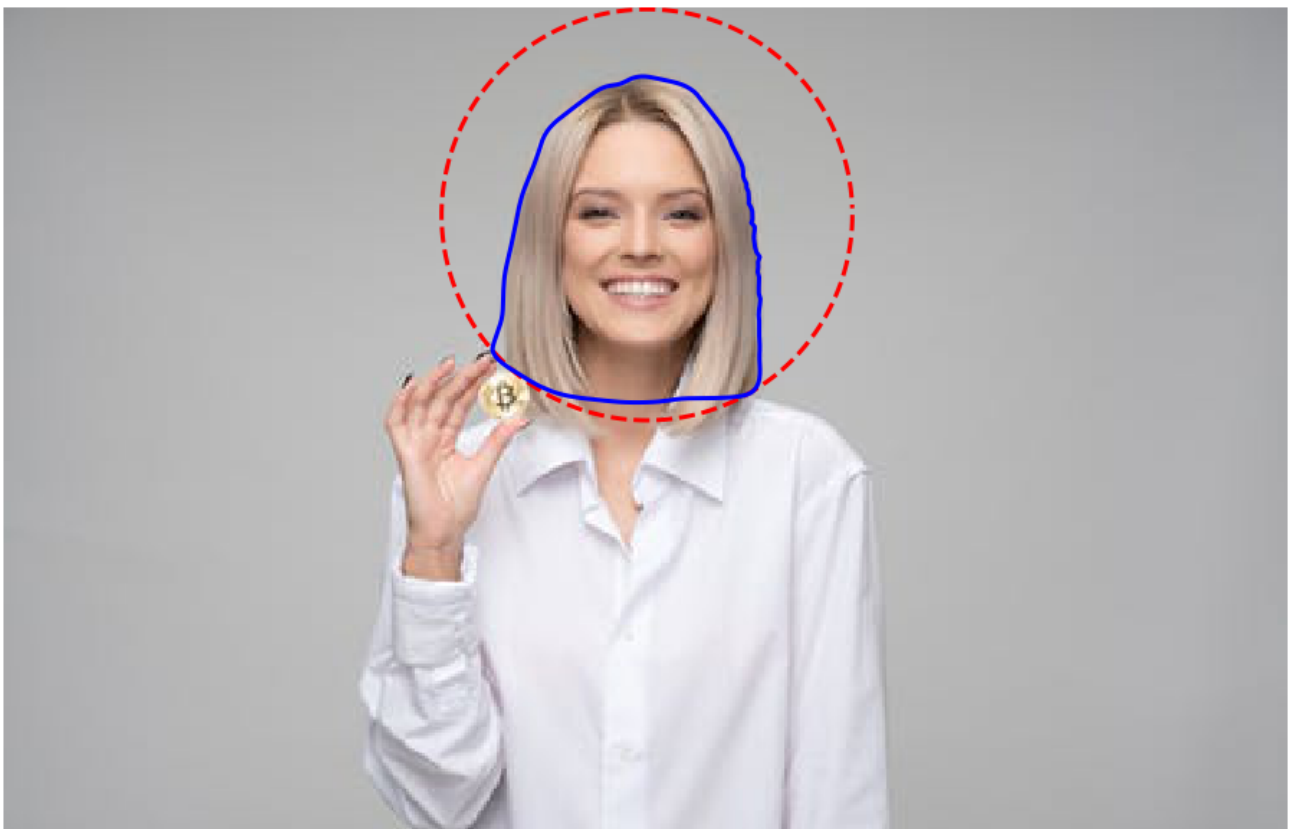
The algorithm then segments the face of a person from the rest of an image by fitting a closed curve to the edges of the face.

```
snake = seg.active_contour(image_gray, points)

fig, ax = image_show(image)
ax.plot(points[:, 0], points[:, 1], '--r', lw=3)
ax.plot(snake[:, 0], snake[:, 1], '-b', lw=3);
```



We can tweak the parameters called `alpha` and `beta`. Higher values of alpha will make this snake contract faster while beta makes the snake smoother.

```
snake = seg.active_contour(image_gray, points,alpha=0.06,beta=0.3)

fig, ax = image_show(image)
ax.plot(points[:, 0], points[:, 1], '--r', lw=3)
ax.plot(snake[:, 0], snake[:, 1], '-b', lw=3);
```



Random walker segmentation

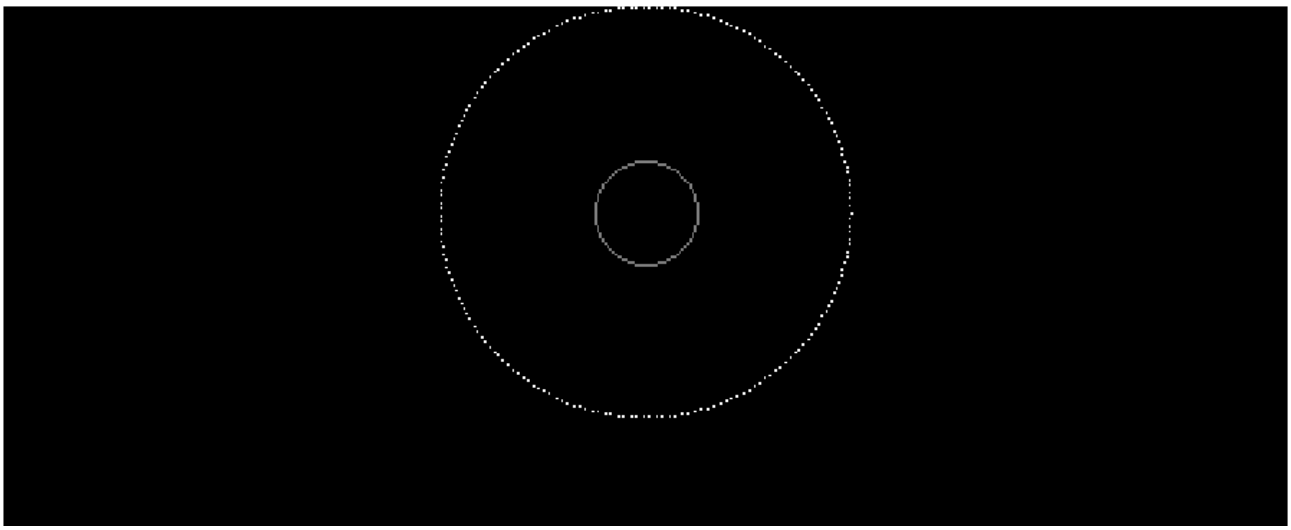
In this method, a user interactively labels a small number of pixels which are known as **labels**. Each unlabeled pixel is then imagined to release a random walker and one can then determine the probability of a random walker starting at each unlabeled pixel and reaching one of the prelabeled pixels. By assigning each pixel to the label for which the greatest probability is calculated, high-quality image segmentation may be obtained. Read the Reference paper here.

We will re-use the seed values from our previous example here. We could have done different initializations but for simplicity let's stick to circles.

```
image_labels = np.zeros(image_gray.shape, dtype=np.uint8)
```

The random walker algorithm expects a label image as input. So we will have the bigger circle that encompasses the person's entire face and another smaller circle near the middle of the face.

```
indices = draw.circle_perimeter(80, 250,20)#from here

image_labels[indices] = 1
image_labels[points[:, 1].astype(np.int), points[:,
0].astype(np.int)] = 2

image_show(image_labels);
```



Now, let's use Random Walker and see what happens.

```
image_segmented = seg.random_walker(image_gray, image_labels)

# Check our results
fig, ax = image_show(image_gray)
ax.imshow(image_segmented == 1, alpha=0.3);
```

It doesn't look like it's grabbing edges as we wanted. To resolve this situation we can tune in the beta parameter until we get the desired results. After several attempts, a value of 3000 works reasonably well.

```
image_segmented = seg.random_walker(image_gray, image_labels, beta =
3000)

# Check our results
fig, ax = image_show(image_gray)
ax.imshow(image_segmented == 1, alpha=0.3);
```

That's all for Supervised Segmentation where we had to provide certain inputs and also had to tweak certain parameters. However, it is not always possible to have a human looking at an image and then deciding what inputs to give or where to start from. Fortunately, for those situations, we have Unsupervised segmentation techniques.

. . .

Unsupervised segmentation

Unsupervised segmentation requires no prior knowledge. Consider an image that is so large that it is not feasible to consider all pixels simultaneously. So in such cases, Unsupervised segmentation can breakdown the image into several sub-regions, so

instead of millions of pixels, you have tens to hundreds of regions. Let's look at two such algorithms:

SLIC(Simple Linear Iterative Clustering)

SLIC algorithm actually uses a machine learning algorithm called **K-Means** under the hood. It takes in all the pixel values of the image and tries to separate them out into the given number of sub-regions. Read the Reference Paper here.

SLIC works in color so we will use the original image.

```
image_slic = seg.slic(image,n_segments=155)
```

All we're doing is just setting each sub-image or sub-region that we have found, to the average of that region which makes it look less like a patchwork of randomly assigned colors and more like an image that has been decomposed into areas that are kind of similar.

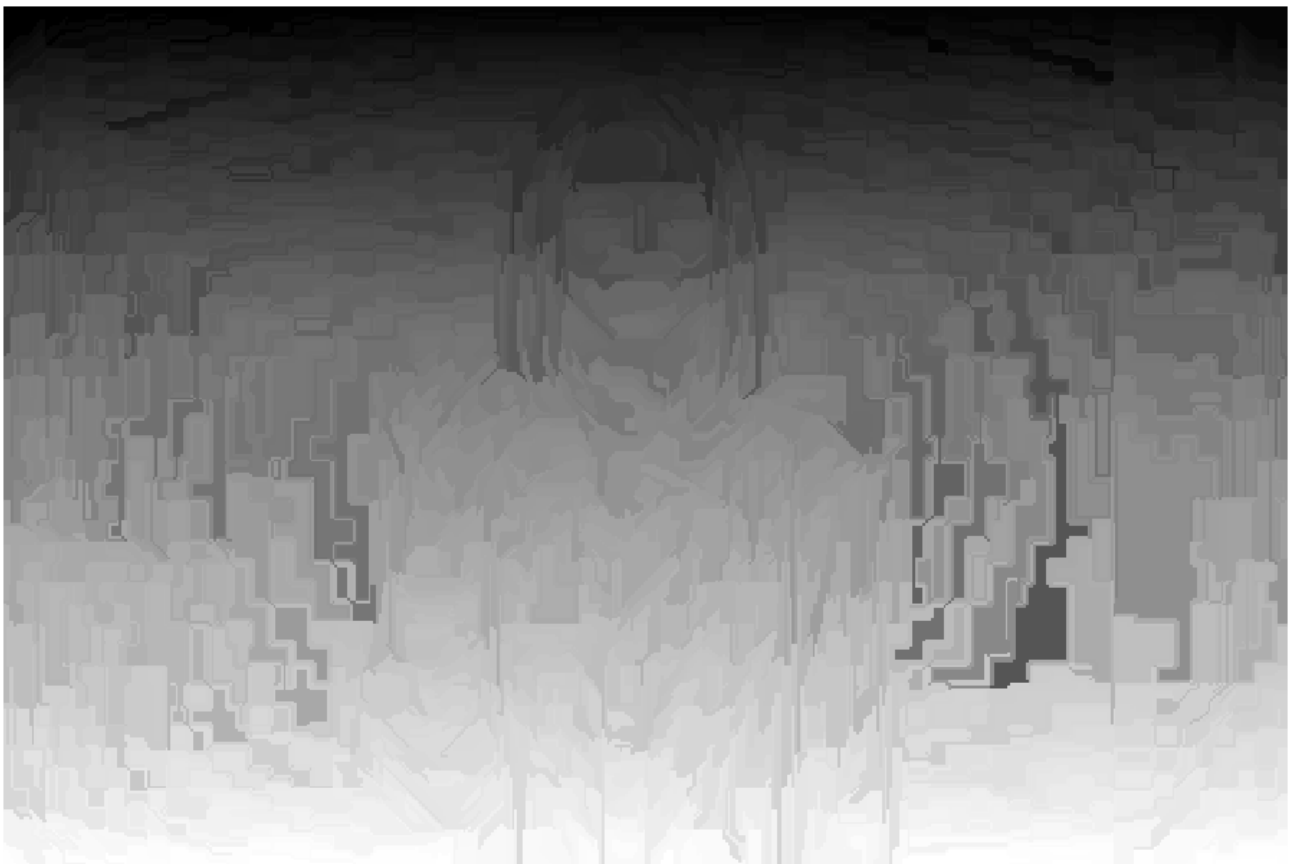
```
# label2rgb replaces each discrete label with the average interior
color
image_show(color.label2rgb(image_slic, image, kind='avg'));
```

We've reduced this image from 512*512 = 262,000 pixels down to 155 regions.

Felzenszwalb

This algorithm also uses a machine learning algorithm called **minimum-spanning tree clustering** under the hood. Felzenszwaib doesn't tell us the exact number of clusters that the image will be partitioned into. It's going to run and generate as many clusters as it thinks is appropriate for that
given scale or zoom factor on the image. The Reference Paper can be accessed here.

```
image_felzenszwalb = seg.felzenszwalb(image)
image_show(image_felzenszwalb);
```



These are a lot of regions. Let's calculate the number of unique regions.

```
np.unique(image_felzenszwalb).size
3368
```

Now let's recolour them using the region average just as we did in the SLIC algorithm.

```
image_felzenszwalb_colored = color.label2rgb(image_felzenszwalb,
image, kind='avg')

image_show(image_felzenszwalb_colored);
```

Now we get reasonable smaller regions. If we wanted still fewer regions, we could change the `scale` parameter or start here and combine them. This approach is sometimes called **over-segmentation**.

This almost looks more like a posterized image which is essentially just a reduction in the number of colours. To combine them again, you can use the **Region Adjacency Graph(RAG)** but that's beyond the scope of this article.

. . .

Conclusion

Image segmentation is a very important image processing step. It is an active area of research with applications ranging from computer vision to medical imagery to traffic and video surveillance. Python provides a robust library in the form of scikit-image having a large number of algorithms for image processing. It is available free of charge and free of restriction having an active community behind it. Have a look at their documentation to learn more about the library and its use cases.

. . .

References:

Scikit image documentation

Machine Learning | Artificial Intelligence | Image Processing | Python | Towards Data Science