Previously, on How to get started with Tesseract, I gave you a practical quick-start tutorial on Tesseract using Python. It is a pretty simple overview, but it should help you get started with Tesseract and clear some hurdles that I faced when I was in your shoes. Now, I’m keen on showing you a few more tricks and stuff you can do with Tesseract and OpenCV to improve your overall accuracy.

**Where did we leave off last time?**

In the previous story, I didn’t bother going into details for the most part. But if you liked the first story, here comes the sequel! So where did we leave off?

Ah, we had a brief overview of rescaling, noise removal, and binarization. Now, it’s time to get down to details and show you a few settings you can play with.

**Rescaling**

The images that are rescaled are either shrunk or enlarged. If you’re interested in shrinking your image, **INTER\_AREA** is the way to go for you. (Btw, the parameters *fx* and *fy* denote the scaling factor in the function below.)

img = cv2.resize(img, None, fx=0.5, fy=0.5, interpolation=cv2.INTER\_AREA)

On the other hand, as in most cases, you may need to scale your image to a larger size to recognize small characters. In this case, **INTER\_CUBIC** generally performs better than other alternatives, though it’s also slower than others.

img = cv2.resize(img, None, fx=2, fy=2, interpolation=cv2.INTER\_CUBIC)

If you’d like to trade off some of your image quality for faster performance, you may want to try **INTER\_LINEAR** for enlarging images.

img = cv2.resize(img, None, fx=2, fy=2, interpolation=cv2.INTER\_LINEAR)

**Blurring**

It’s worth mentioning that there are a few blur filters available in the [OpenCV library](https://docs.opencv.org/3.4.0/d4/d13/tutorial_py_filtering.html" \t "_blank). Image blurring is usually achieved by convolving the image with a low-pass filter kernel. While filters are usually used to blur the image or to reduce noise, there are a few differences between them.

**1. Averaging**

After convolving an image with a normalized box filter, this simply takes the average of all the pixels under the kernel area and replaces the central element. It’s pretty self-explanatory, I guess.

img = cv.blur(img,(5,5))

**2. Gaussian blurring**

This works in a similar fashion to Averaging, but it uses Gaussian kernel, instead of a normalized box filter, for convolution. Here, the dimensions of the kernel and standard deviations in both directions can be determined independently. Gaussian blurring is very useful for removing — guess what? — gaussian noise from the image. On the contrary,gaussian blurring does not preserve the edges in the input.

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

**3. Median blurring**

The central element in the kernel area is replaced with the median of all the pixels under the kernel. Particularly, this outperforms other blurring methods in removing salt-and-pepper noise in the images.

Median blurring is a non-linear filter. Unlike linear filters, median blurring replaces the pixel values with the median value available in the neighborhood values. So, median blurring preserves edges as the median value must be the value of one of neighboring pixels.

img = cv2.medianBlur(img, 3)

**4. Bilateral filtering**

Speaking of keeping edges sharp, bilateral filtering is quite useful for removing the noise without smoothing the edges. Similar to gaussian blurring, bilateral filtering also uses a gaussian filter to find the gaussian weighted average in the neighborhood. However, it also takes pixel difference into account while blurring the nearby pixels.

Thus, it ensures only those pixels with similar intensity to the central pixel are blurred, whereas the pixels with distinct pixel values are not blurred. In doing so, the edges that have larger intensity variation, so-called edges, are preserved.

img = cv.bilateralFilter(img,9,75,75)

Overall, if you are interested in preserving the edges, go with median blurring or bilateral filtering. On the contrary, gaussian blurring is likely to be faster than median blurring. Due to its computational complexity, bilateral filtering is the slowest of all methods.

Again, you do you.

**Image Thresholding**

There’s not a single image thresholding method that fits all types of documents. In reality, all filters perform differently on varying images. For instance, while some filters successfully binarize some images, they may fail to binarize others. Likewise, some filters may work well with those images that other filters cannot binarize well.

I’ll try to cover the basics here, though I do recommend that you read the official documentation of OpenCV on Image Thresholding for more information and the theory behind it.

**1. Simple Threshold**

You might recall a friend of yours giving you some advice about your life by saying “things are not always black and white”. Well, for a simple threshold, things are pretty straight-forward.

cv.threshold(img,127,255,cv.THRESH\_BINARY)

First, you pick a threshold value, say 127. If the pixel value is greater than the threshold, it becomes black. If less, it becomes white. OpenCV provides us with different types of thresholding methods that can be passed as the fourth parameter. I often use binary threshold for most tasks, but for other thresholding methods you may visit the official documentation.

**2. Adaptive Threshold**

Rather than setting a one global threshold value, we let the algorithm calculate the threshold for small regions of the image. Thus, we end up having various threshold values for different regions of the image, which is great!

cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, 31, 2)

There are two adaptive methods for calculating the threshold value. While **Adaptive Thresh Mean** returns the mean of the neighborhood area, **Adaptive Gaussian Mean** calculates the weighted sum of the neighborhood values.

We’ve got two more parameters that determine the size of the neighborhood area and the constant value that is subtracted from the result: the fifth and sixth parameters, respectively.

**3. Otsu’s Threshold**

This method particularly works well with **bimodal images**, which is an image whose histogram has two peaks. If this is the case, we might be keen on picking a threshold value between these peaks. This is what Otsu’s Binarization actually does, though.

cv2.threshold(img, 0, 255, cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)[1]

It’s pretty useful for some cases. But it may fail to binarize images that are not bimodal. So, please take this filter with a grain of salt.

**Types of thresholding**

You might have already noticed there is a parameter, or in some cases a combination of a few parameters, that are passed as arguments to determine the type of thresholding, such as THRESH\_BINARY. I’m not going into the detail here now, as it is explained clearly in the official documentation.

**What next?**

So far, we’ve discussed some of the techniques of image pre-processing. You might wonder when exactly you’re going to get your hands dirty. Well, the time has come. Before you get back to your favorite Python IDE — mine is [PyCharm](https://www.jetbrains.com/pycharm/" \t "_blank), btw — I’m going to show you few lines of code that will save you some time while trying to find which combination of filters and image manipulations work well with your documents.

Let’s start by defining a switcher function that holds a few combinations of thresholding filters and blurring methods. Once you get the idea, you could also add more filters, incorporating other image pre-processing methods like rescaling into your filter set.

Here I’ve created 20 different combinations of image thresholding methods, blurring methods, and kernel sizes. The switcher function, *apply\_threshold*, takes two arguments, namely OpenCV image and an integer that denotes the filter. Likewise, since this function returns the OpenCV image as a result, it could easily be integrated into our *get\_string* function from the previous post.

def apply\_threshold(img, argument):  
 switcher = {  
 1: cv2.threshold(cv2.GaussianBlur(img, (9, 9), 0), 0, 255, cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)[1],  
 2: cv2.threshold(cv2.GaussianBlur(img, (7, 7), 0), 0, 255, cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)[1],  
 3: cv2.threshold(cv2.GaussianBlur(img, (5, 5), 0), 0, 255, cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)[1],

**...**

18: cv2.adaptiveThreshold(cv2.medianBlur(img, 7), 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, 31, 2),  
 19: cv2.adaptiveThreshold(cv2.medianBlur(img, 5), 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, 31, 2),  
 20: cv2.adaptiveThreshold(cv2.medianBlur(img, 3), 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, 31, 2)  
 }  
 return switcher.get(argument, "Invalid method")

And, here it comes.

def get\_string(img\_path, method):  
 # Read image using opencv  
 img = cv2.imread(img\_path)  
  
 # Extract the file name without the file extension  
 file\_name = os.path.basename(img\_path).split('.')[0]  
 file\_name = file\_name.split()[0]  
  
 # Create a directory for outputs  
 output\_path = os.path.join(output\_dir, file\_name)  
 if not os.path.exists(output\_path):  
 os.makedirs(output\_path)

# Rescale the image, if needed.  
 img = cv2.resize(img, None, fx=1.5, fy=1.5, interpolation=cv2.INTER\_CUBIC)

# Convert to gray  
 img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
  
 # Apply dilation and erosion to remove some noise  
 kernel = np.ones((1, 1), np.uint8)  
 img = cv2.dilate(img, kernel, iterations=1)  
 img = cv2.erode(img, kernel, iterations=1)

**# Apply threshold to get image with only black and white  
 img = apply\_threshold(img, method)**

# Save the filtered image in the output directory  
 save\_path = os.path.join(output\_path, file\_name + "\_filter\_" + str(method) + ".jpg")  
 cv2.imwrite(save\_path, img)  
  
 # Recognize text with tesseract for python  
 result = pytesseract.image\_to\_string(img, lang="eng")

return result

**Last words**

Now, all we need to do is to write a simple for loop that iterates over the input directory to collect images and applies each filter on the images gathered. I prefer to use *glob*, or *os*, for collecting images from directories, and *argparse*for passing arguments via terminal, like any other sane person would do.

Here I’ve done pretty much the same thing as in my [gist](https://gist.github.com/bkaankuguoglu/111f9f5e0c30b5f57d7c5338d6dcb6fc" \t "_blank), if you’d like have a look at it. However, feel free to use the tools you feel comfortable with.

So far, I’ve tried to cover a few useful image pre-processing concepts and implementations, though it’s probably just the tip of the iceberg. I don’t know how much “leisure time” I’m going to have in the upcoming weeks, so, I can’t give you a specific time frame for publishing my next post. However, I’m considering adding at least one more part to this series that explains a few things I left out, such as rotation and de-skewing on images.

Until then, best bet is to just keep your wits about you and continue to look for signs.[\*](https://www.youtube.com/watch?v=B_CHjYoqPUU)