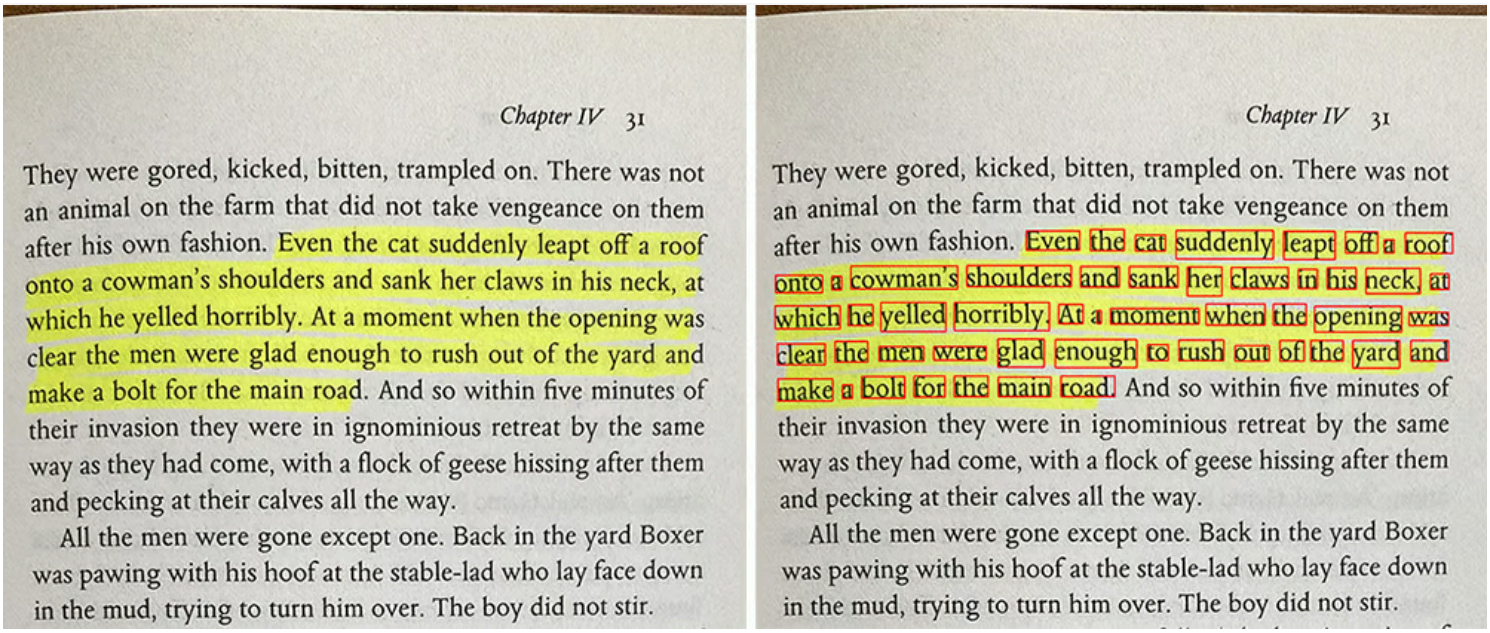


Chapter IV 31

They were gored, kicked, bitten, trampled on. There was not an animal on the farm that did not take vengeance on them after his own fashion. Even the cat suddenly leapt off a roof onto a cowman's shoulders and sank her claws in his neck, at which he yelled horribly. At a moment when the opening was clear the men were glad enough to rush out of the yard and make a bolt for the main road. And so within five minutes of their invasion they were in ignominious retreat by the same way as they had come, with a flock of geese hissing after them and pecking at their calves all the way.

All the men were gone except one. Back in the yard Boxer was pawing with his hoof at the stable-lad who lay face down in the mud, trying to turn him over. The boy did not stir.

Chris Cook
Posted on Feb 22, 2022 • Updated on Mar 14, 2023

Extract Highlighted Text from a Book using Python

#python #ocr #opencv #tesseract

I'm old fashioned when it comes to reading. Even though I prefer the digital equivalent in almost every other aspect of life, when it comes to reading, I almost always prefer paper books to e-books, PDFs, or other electronic media. I enjoy the haptic feedback of reading a book and the visual progress you see each time you turn a page as the remaining part of the book diminishes more and more.

However, taking notes and creating bookmarks with a paper book is quite tedious. Of course, you can highlight certain paragraphs and take a picture of that page to save it in your notes' app of choice. That's literally what I do. But then how do you find this particular piece of information again? I was inspired by this article by Shaham in which he described a digital highlighter that recognizes and extracts highlighted text on a book page. Fortunately, he didn't share much code, and the web application is no longer live. Rather than using his work, I took on the challenge of rebuilding the

algorithm and implementing it myself. I'm not familiar with any of the technologies used, so the real goal was to learn and have fun.

Some More Words

Before we begin, I would like to briefly explain the basic structure of this post and a few words about the technologies involved. The structure of this post resembles the path I followed when I was working on this problem and its implementation. First, to extract all the text on a book page, regardless of whether it is highlighted or not. Second, to find the outlines of the yellow highlighted areas. Finally, these two sections are merged to extract only the text that lies within the outlines of a highlighted area and is therefore highlighted text. I added some code snippets to each section, but mainly as reference for those of us that prefer to read code over words. However, I'm going to focus on explaining the concepts behind the implementation as these are transferable to other technologies.

Optical Character Recognition

Optical Character Recognition (OCR) is a process to extract written or printed text from a document - such as an image - and to convert it into digital text that can be used for further processing, e.g. to index this text in a database and access it via a search engine. Some may remember the effort of Google to digitize every book on the planet and make it available via their Google Books search, or Project Guttenberg which digitizes and provides public domain books.

I'm going to use the Tesseract OCR engine and library, and its Python wrapper PyTesseract for text extraction. But there are numerous libraries out there to extract text from an image. In a real world application I would probably use cloud services from AWS, Google or Microsoft to handle this task.

Computer Vision

Computer Vision (CV) is a broad description of various methods to programmatically analyze, process and understand an image.
It's often used as a pre-processing or post-processing step to improve the result of the actual intended use case, for example the application of OCR.

I'm going to use the OpenCV library and its Python interface. OpenCV encompasses many functions for object detection in image and video, machine learning, image

processing and many more.

Animal Farm

I have used pages from George Orwell's Animal Farm for my examples below. Since 2021, this book has entered the public domain and therefore content from this book can be shared without infringement.

Extract All Text

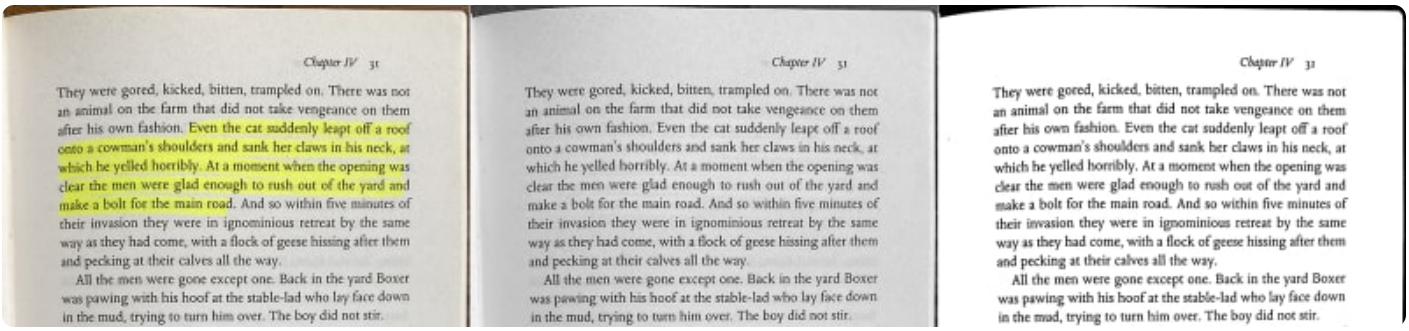
As the first step, we are going to start with OCR and simply extract all text from an image without respecting any highlights. In order for the OCR algorithm to work properly, we have to provide a binary image. To convert our original image into a binary, we are going to pre-process the image via OpenCV Thresholding. There are some other methods to improve the quality as described in the documentation.

Thresholding

First, we have to grayscale the image with a simple color space conversion from RGB to Gray. This is the prerequisite for the actual operation. Thresholding is an operation to separate an image into two distinct layers of pixels, i.e. foreground and background. The result is a binary image with only black and white pixels and no color shades in between. The foreground layer contains the white pixels that carry/hold the actual information, e.g. the words and letters. I'm using Otsu's algorithm to automatically determine the optimal threshold value.

```
def threshold_image(img_src):
    """Grayscale image and apply Otsu's threshold"""
    # Grayscale
    img_gray = cv2.cvtColor(img_src, cv2.COLOR_BGR2GRAY)
    # Binarisation and Otsu's threshold
    _, img_thresh = cv2.threshold(
        img_gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

    return img_thresh, img_gray
```

The following images show the original image on the left, the grayscale image in the middle, and the binary image as thresholding result on the right.



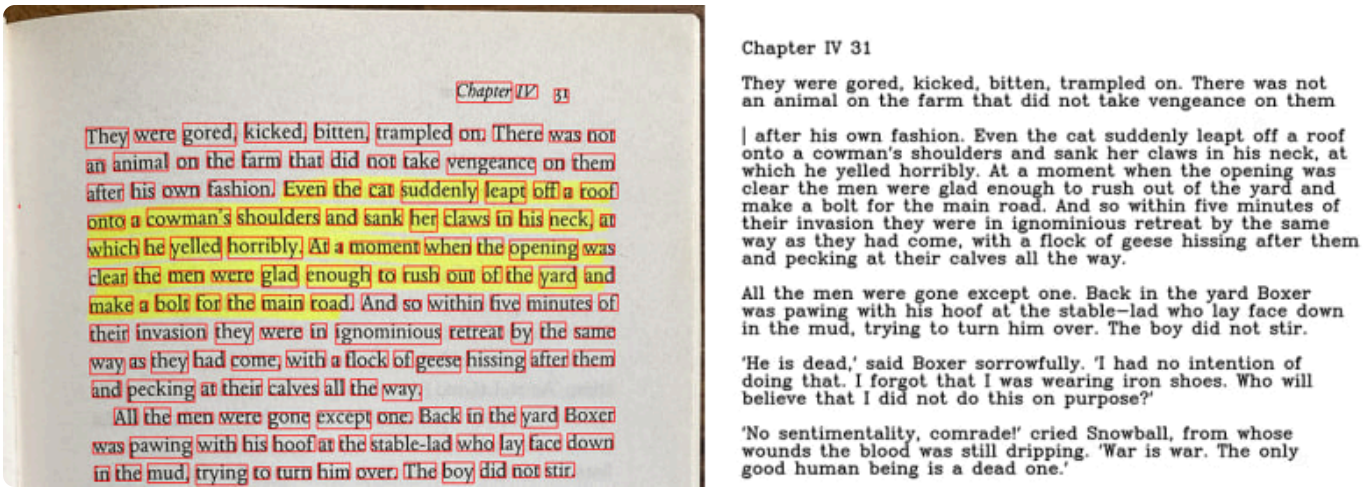
Comparison of original, grayscale and binary image

OCR

Then we can continue with the extraction of text. The binary (thresholded) image is the input for OCR. Tesseract, the OCR engine, runs its language models on the supplied image and returns recognized strings (i.e. letters, words, digits) and complimentary metadata like bounding boxes, confidence levels, and page layout information. A bounding box is the physical position and size - these are x/y coordinates plus width and height - of an object on an image.

```
def extract_all(img_src):
    # Extract all text as one string
    string_ocr = pytesseract.image_to_string(
        img_thresh, lang='eng', config='--psm 6')
    # Extract all text and meta data as dictionary
    data_ocr = pytesseract.image_to_data(
        img_src, lang='eng', config='--psm 6', output_type=Output.DICT)
    # Copy source image to draw rectangles
    img_result = img_src.copy()

    # Iterate through all words
    for i in range(len(data_ocr['text'])):
        # Skip other levels than 5 (word)
        if data_ocr['level'][i] != Levels.WORD:
            continue
        # Get bounding box position and size of word
        (x, y, w, h) = (data_ocr['left'][i], data_ocr['top']
                        [i], data_ocr['width'][i], data_ocr['height'][i])
        # Draw rectangle for word bounding box
        cv2.rectangle(img_result, (x, y), (x + w, y + h), (0,0,255), 2)

    return img_result
```

In order to visualize the OCR result, I added red rectangles for the bounding boxes of recognized string on the original image. Next to this image is the actual text extraction from OCR. I printed the returned string from Tesseract on an image to be able to compare it directly with the original. The result of the text extraction is quite decent, although there are some errors.



Word bounding boxes and extracted text

Find Highlighted Areas

To extract only the highlighted text and discard the rest, we first need to find the yellow highlighted areas on a page. We first look for areas with similar color and then adjust the image so that these areas are related. Since this part is independent of the previous one, we start again with the original image.

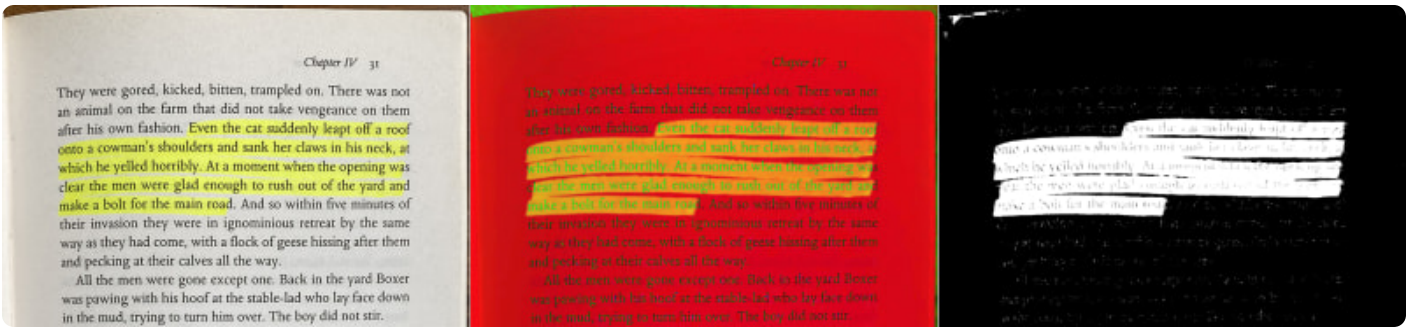
Color Segmentation

First, we perform a color segmentation. This operation, similar to thresholding, separates a specific color - in this case yellow - from an image into its own layer. The result is a color mask, a binary image that can be used to cover an image so that certain pixels (i.e. highlighted) show through and others are hidden. Segmentation is performed in HSV color space, so we need to convert the original image from RGB to HSV first. The color is defined as a range with an upper and lower limit, since the actual color may vary due to various conditions such as light, shadow, or exposure.

```
def mask_image(img_src, lower, upper):
    """Convert image from RGB to HSV and create a mask for given lower and uppe
    # RGB to HSV color space conversion
    img_hsv = cv2.cvtColor(img_src, cv2.COLOR_BGR2HSV)
    hsv_lower = np.array(lower, np.uint8)  # Lower HSV value
```

```
    hsv_upper = np.array(upper, np.uint8)  # Upper HSV value

    # Color segmentation with lower and upper threshold ranges to obtain a bina
    img_mask = cv2.inRange(img_hsv, hsv_lower, hsv_upper)

    return img_mask, img_hsv
```

Here you see the original image on the left, the HSV converted in the middle, and the final binary image mask on the right:



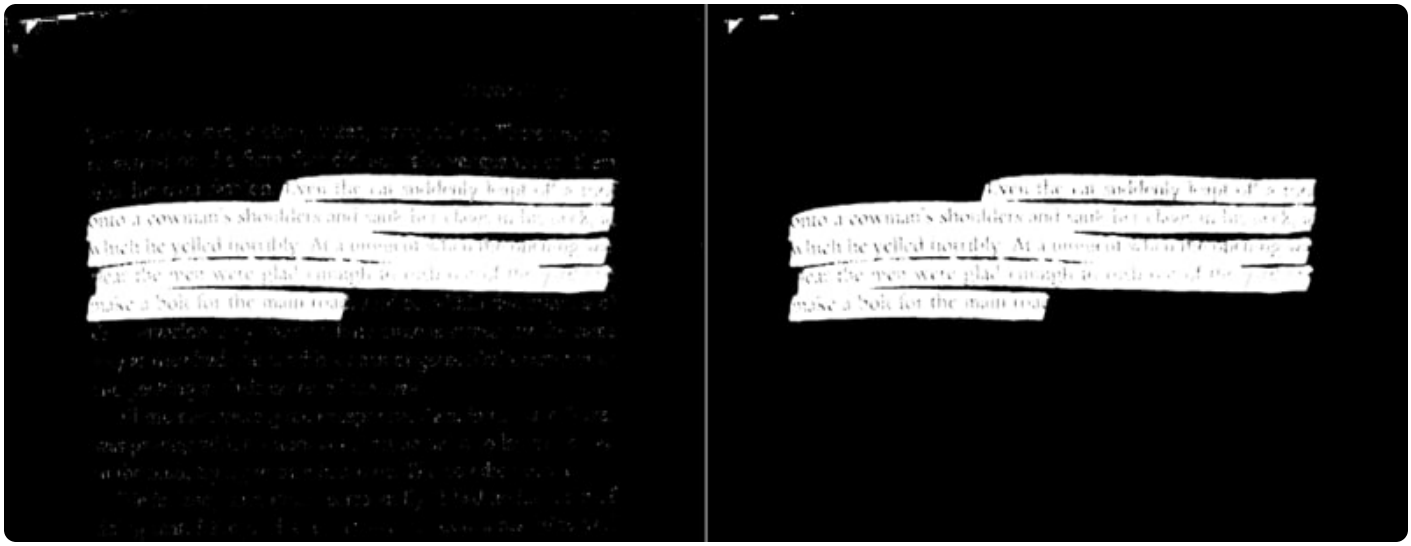
Comparison of original, HSV converted and binary image

Noise Reduction

Then we continue to reduce the noise from the image, also called image denoising. Noise are those seemingly random pixels. It usually occurs in the image sensor when a photo is taken, but in our case it probably comes from areas that appeared yellow-ish and therefore, fell in our color range. These pixels could later affects our program in distinguishing whether a word is highlighted or not. We use a morphological transformation remove these small points and smooth the edges. However, depending on the original image and the conditions under which it was captured, additional steps may be required to remove the noise.

```
def denoise_image(img_src):
    """Denoise image with a morphological transformation."""

    # Morphological transformations to remove small noise
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
    img_denoised = cv2.morphologyEx(
        img_src, cv2.MORPH_OPEN, kernel, iterations=1)

    return img_denoised
```

Here is a comparison of the image mask before and after image denoising, even though the noise might be difficult to see (small white dots). The effect may appear trivial, but it improves the subsequent contour detection enormously.



Comparison of image before and after denoising

Extract Highlighted Text

So far, we have been able to extract the text from an image and separate the highlighted areas from the rest of the image. Now we need to put these two parts together in order to extract only the highlighted text. This sounds simple, but there are some obstacles along the way. I will go through a few attempts.

Apply Image Mask

The first, most obvious idea was to use the image mask created in the previous chapter and use it to hide all non-highlighted areas. Technically, we will perform a bitwise AND operation between the original image and the image mask.

```
def apply_mask(img_src, img_mask):
    """Apply bitwise conjunction of source image and image mask."""

    img_result = cv2.bitwise_and(img_src, img_src, mask=img_mask)

    return img_result
```

The following image illustrates what the operation does.
I have applied the image mask to the original image and to the thresholded (binary)

image to make the effect clearer. For OCR we would only need to apply the mask to the binary image.

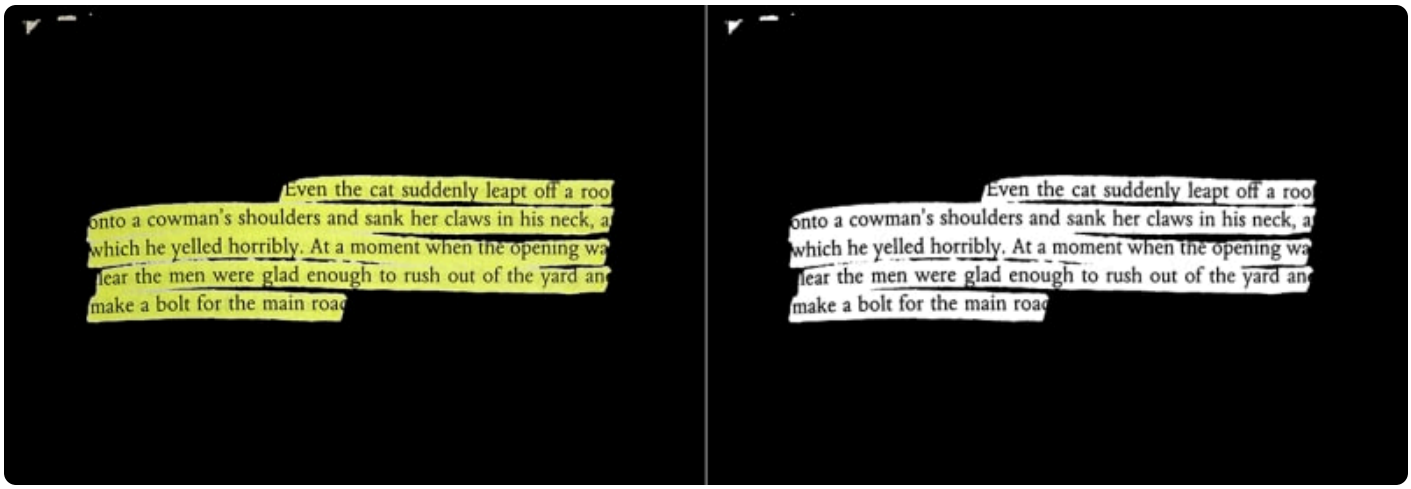


Image mask bitwise AND on original and binary image

Even though this option looks reasonable, if you were to apply OCR to this masked image, you would quickly find that not all words can be recognized because certain words are not properly highlighted: a letter might be missing from the beginning or end, or a word might not be clearly marked.

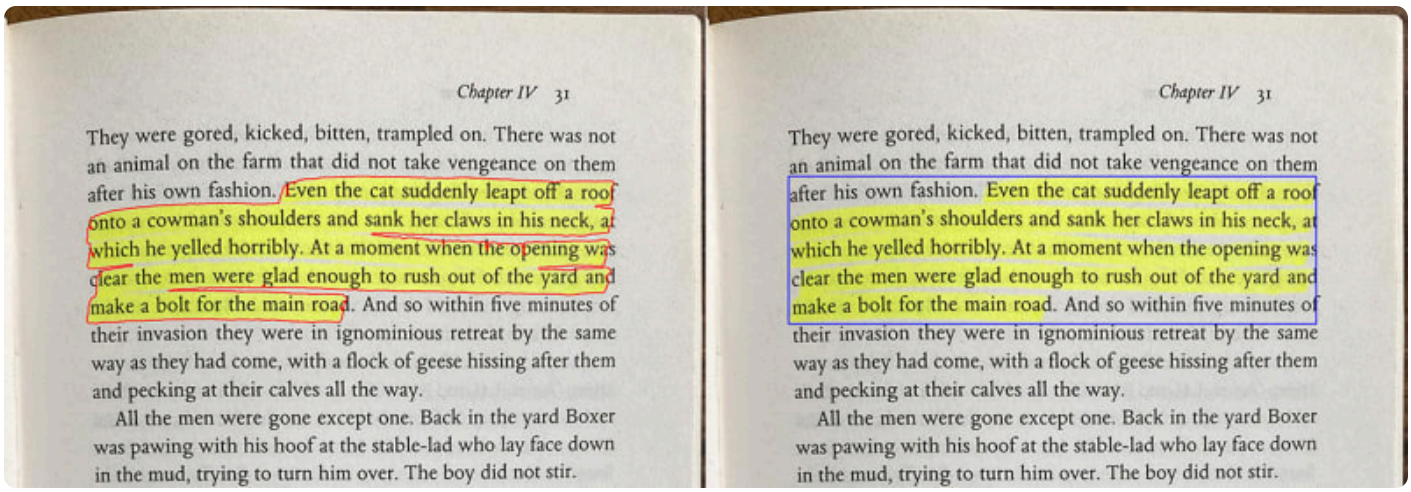
Contour Bounding Box

My second attempt was to find the outlines of the highlighted areas using the contour detection on the image mask. Each contour has a bounding box (called a bounding rect in OpenCV) that describes its position and size. Then we can crop the image to the size of the bounding box and apply OCR only to the relevant area of the image.

```
def draw_contour_boundings(img_src, img_mask, threshold_area=400):
    """Draw contour bounding and contour bounding box"""
    # Contour detection
    contours, hierarchy, = cv2.findContours(
        img_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Create two copies of source image
    img_contour = img_src.copy()
    img_box = img_src.copy()

    for idx, c in enumerate(contours):
        # Skip small contours because its probably noise
        if  cv2.contourArea(c) < threshold_area:
            continue
```

```
        # Draw contour in red
        cv2.drawContours(img_contour, contours, idx, (0, 0, 255), 2, cv2.LINE_4

        # Get bounding box position and size of contour
        x, y, w, h = cv2.boundingRect(c)
        # Draw bounding box in blue
        cv2.rectangle(img_box, (x, y), (x + w, y + h), (255, 0, 0), 2, cv2.LINE

    return img_contour, img_box
```

Here is the original image with the red outline on the left and the blue bounding box on the right:



Comparison of contour outline vs. contour bounding box

Obviously, an issue with this approach is that the bounding box of the highlighted area is a rectangle, which in most cases has a larger area than the actual highlighted area. This means that the rectangle may include words that are not highlighted, such as the highlighting starting in the middle of a sentence or ending on a new line.

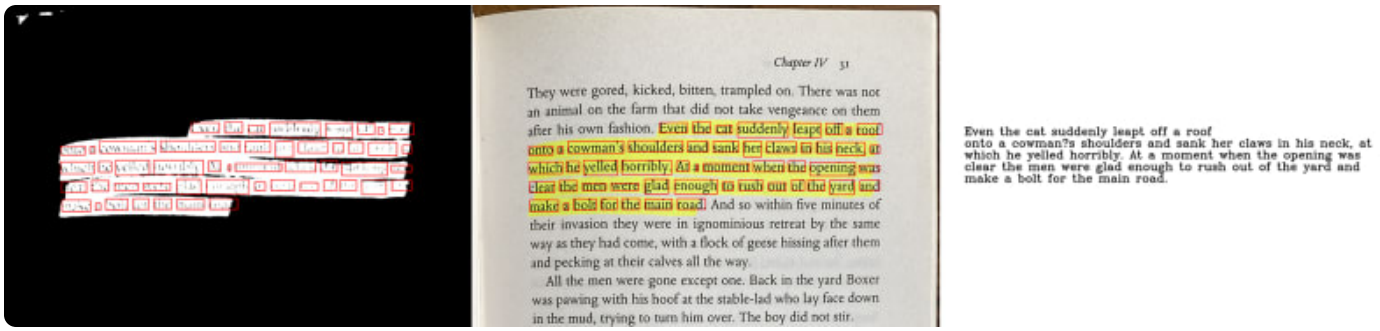
Calculate Highlighted Area Foreach Word

This approach builds on the previous two ideas. However, instead of extracting only highlighted words, we will OCR every word on a page and then decide word by word whether it is highlighted or not.
Going back to the first chapter, we applied OCR to the entire page. The engine returned each recognized string along with its bounding box - the position and size - on the image. From here, it's easy to calculate the area of the bounding box, as it's basically just a rectangle with two points: `x1/y1` and `x2/y2`. With this rectangle, also called

Region of Interest or ROI, we select exactly the same area on the image mask. Now, in order for a word to be considered "highlighted", we need to determine how much of its area described by its bounding box (rectangle) is white compared to its total area. This ratio can be expressed as a percentage as $\frac{white\ pixels}{total\ pixels} \times 100$ and must be determined for each word. Finally, we can simply go through all the words and check if the highlighted area percentage exceeds a certain threshold, e.g. 25%. If so, the area is considered highlighted and added to the output, otherwise it is discarded.

```
def find_highlighted_words(img_mask, data_ocr, threshold_percentage=25):
    """Find highlighted words by calculating how much of the words area contain

    # Initiliaze new column for highlight indicator
    data_ocr['highlighted'] = [False] * len(data_ocr['text'])

    for i in range(len(data_ocr['text'])):
        # Get bounding box position and size of word
        (x, y, w, h) = (data_ocr['left'][i], data_ocr['top']
                        [i], data_ocr['width'][i], data_ocr['height'][i])
        # Calculate threshold number of pixels for the area of the bounding box
        rect_threshold = (w * h * threshold_percentage) / 100
        # Select region of interest from image mask
        img_roi = img_mask[y:y+h, x:x+w]
        # Count white pixels in ROI
        count = cv2.countNonZero(img_roi)
        # Set word as highlighted if its white pixels exceeds the threshold val
        if count > rect_threshold:
            data_ocr['highlighted'][i] = True

    return data_ocr
```

I have tried to illustrate this concept with the following images. The left one is the binary image we use for OCR, and the green rectangles are the word boundaries recognized by OCR. On the right side, I copied the same rectangles onto the image mask to emphasize the position of each word in relation to the highlighted (white) area. Some words are either inside the white (highlighted) area or outside it in the black area. And few are somewhere in the middle, as they overlap with white and black. For each rectangle (word), we calculate its area which equals the total number of pixels. Each individual pixel has a color, black or white, which is represented digitally as a number of 0 or 1. And with OpenCV we can easily count all pixels that are not zero (i.e. white pixels) in an image or in this case in a region (bounding box or rectangle).



Comparison of word bounding boxes on image mask and original image, and highlighted extracted text

Conclusion

That's about it. My intention was to play around with Python, OCR and computer vision and learn something. I'm sure there are many other, probably better, ways to implement something like this. If anyone has any feedback - be it criticism, ideas or suggestions - please let me know. I am very interested in any improvements to my approach or code. If you're interested in testing it yourself, you will soon find all the code with instructions on how to run it locally in my GitHub repository. I am preparing it right now to make it available to the public.

I hope you found this post helpful. If you have any questions or comments, feel free to leave them below. If you'd like to connect with me, you can find me on LinkedIn or GitHub. Thanks for reading!

Top comments (8)

UponTheSky • May 19 '23

This is such a great project!

Chris Cook • May 20 '23

Thank you! :) I thinking about developing it further into a web application so it's easy to use in a browser.

Nik • Feb 23 '23

That's just incredible! I guess I will adapt this in my next project. Thanks

Ross Watson • Feb 23 '23

Did you productize this yet?

Chris Cook • Feb 23 '23

No, I got some ideas in mind but haven't taken the time to develop it further. A mobile app to quickly capture highlighted text would be nice.

Corentin • Mar 12 '23

Hi, i have an issue with your code. In the function extract_all(), what is the variable Levels ?
Thanks

Chris Cook • Mar 14 '23

The `level` variables describes the layout of the text:

1. page
2. block
3. paragraph
4. line
5. word

You can find a more detailed explanation on this post.

murat • Aug 19 '22

Sir, I want to ask a question if you are avaliable. You have used array in Color Segmentation but i couldn't see the array. Could you help me?

Code of Conduct • Report abuse

Chris Cook

I post things so I don't forget them

WORK
Co-Founder at Flyweight.io

JOINED
May 16, 2021

More from Chris Cook

Automatically Transcribe YouTube Videos with OpenAI Whisper
#python #openai #ai #tutorial

DEV Community

Check out this survey and help us moderate our community by becoming a tag moderator here at DEV.