**Evaluation of the System**

LUO Wenxuan SID:5581504

**1.1 Introduction**

Pytesseract is an Optical Character Recognition (OCR) library for Python that excels at identifying text within images and transforming it into editable data. Built on Google's Tesseract OCR engine, this library provides developers with a straightforward way to incorporate text recognition features into their Python projects. In this assignment, we will preprocess two receipt images using various methods before employing Pytesseract to build a text detection system capable of recognizing words and text in these processed images. By analyzing the output, we can generate an evaluation report on the effectiveness of this system.

This report will be structured into two main sections. The first section will elaborate on my comprehension of the algorithms and techniques utilized in the system. Following that, I will assess the system's efficiency and accuracy, providing a detailed evaluation based on our findings.**1.2 Conceptual Illustration**

Before starting to construct the entire system, I identified four essential libraries that are crucial for completing this task: cv2, numpy, matplotlib, and pytesseract. By downloading these libraries and using import statements such as ‘import’ and ‘from ... import ... as ...’, we can access the functions provided by these libraries, streamlining the development process.

Initially, we can use functions from the matplotlib library to import our original image. To begin, a variable should be assigned to specify the absolute or relative path of the image file. Subsequently, the cv2.imread(path) function can be used to load the image. It is important to note that OpenCV (cv2) loads images in BGR format by default. Therefore, we must use the function cv2.cvtColor(image, cv2.COLOR\_BGR2RGB) to convert the image from BGR to RGB format. This ensures that the image is in the correct color space for further processing. Additionally, we can employ plt.title() to add a custom title to the image and plt.axis('off/on') to control the display of the axis.

Following the image importation, we continue to the important step of pre-processing. This step is particularly necessary when dealing with images that are blurred, noisy, or overly colored. In this context, I have explored four distinct pre-processing techniques. Firstly, the function cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY) can be utilized to convert the image to grayscale, effectively reducing the impact of excessive colors. Secondly, cv2.GaussianBlur() is employed to reduce noise within the image. Thirdly, cv2.threshold() is used to generate a high-contrast image, such as a binary black-and-white image, which is highly effective for text detection purposes. Lastly, the cv2.Canny() function is applied to detect edges, converting pixels above a certain threshold into clear white edges against a black background. This edge detection technique proves to be a valuable pre-processing method.

The next phase involves the primary task of text detection. For this purpose, I utilize the pytesseract.image\_to\_string(image, config=custom\_config) function to convert textual content within images into editable strings. The config=custom\_config parameter is employed to set specific options for Tesseract OCR. This customized configuration comprises two main parameters: OEM (OCR Engine Mode) and PSM (Page Segmentation Mode). These parameters allow for fine-tuning the OCR process to achieve optimal text recognition results.

OEM stands for OCR Engine Mode, and Tesseract has four different OCR engine modes(Rosebrock, 2021):

0: Use the traditional Tesseract OCR engine.

1: Use the LSTM (Long Short-Term Memory) based OCR engine.

2: Use a combination of the traditional Tesseract OCR engine and the LSTM engine.

3: Default mode, automatically select the most suitable engine.

PSM stands for Page Segmentation Mode, and Tesseract has multiple page segmentation modes:

0: Orientation and script detection (OSD) only.

1: Automatic page segmentation with OSD.

2: Automatic page segmentation, but no OSD, or OCR.

3: Fully automatic page segmentation, but no OSD. (Default)

4: Assume a single column of text of variable sizes.

5: Assume a single uniform block of vertically aligned text.

6: Assume a single uniform block of text.

7: Treat the image as a single text line.

8: Treat the image as a single word.

9: Treat the image as a single word in a circle.

10: Treat the image as a single character.

11: Sparse text. Find as much text as possible in no particular order.

12: Sparse text with OSD.

13: Raw line. Treat the image as a single text line, bypassing hacks that are Tesseract-specific.

After trying several combinations, I decided to use ‘oem-3, psm-3’ as my customized set of parameters.

**1.3 Evaluation of the Result**

Figure 1:Original Image

From an efficiency perspective, the average processing times for Pytesseract to handle different types of pre-processed images are as follows: 8 seconds for gray images, 5 seconds for blurred images, 7 seconds for binary images, and 7 seconds for edge-detected images. This data suggests that the cv2.GaussianBlur() method could potentially enhance the speed of the text detection process. The significant reduction in noise achieved by applying Gaussian blur likely allows the system to run more efficiently and quickly compared to other image types.

From an accuracy perspective, the system struggles with simultaneous recognition of both vertical and horizontal texts across these four types of images. Consequently, the performance is notably poor when attempting to recognize vertical text, such as 'SALES INVOICE #12345A' located on the left side of the original images. However, when it comes to horizontal text, the system demonstrates approximately 80% accuracy in gray and binary images. In these two image types, the system tends to recognize smaller characters more accurately than larger ones. Interestingly, the system performs best with blurred images, successfully identifying words that it fails to recognize in gray and binary images, and achieving the highest accuracy overall. In stark contrast, the system performs poorly with edge-detected images, likely due to the pre-processing steps that result in unclear boundaries, making it difficult for the system to accurately detect and recognize text.

**1.4 Prospect**

Eventually, I considered various methods to enhance my system. I could experiment with additional pre-processing techniques for image handling or explore other OCR engines and tools, such as EAST, which offer more efficient and accurate text detection capabilities. Moreover, maybe expand some additional functions like multi-languages detection.