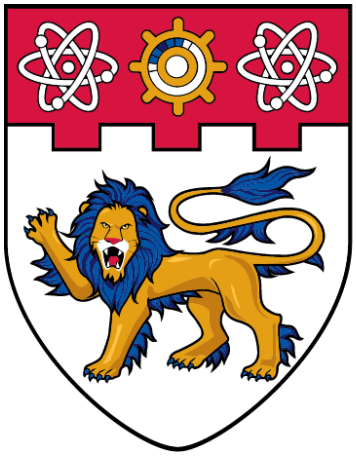


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE4003: Computer Vision

Project: Text Image Segmentation for Optimal Optical Character Recognition

Dwivedee Lakshyajeet (U1822289L)
Mittal Madhav (U1822408H)

School of Computer Science and Engineering

Contributions

Dwivedee Lakshyajeet

1. Code for custom Otsu thresholding
2. Code for OCR accuracy calculation using cosine similarity
3. Code for adaptive Otsu thresholding
4. Code for background removal using variance
5. Code for background removal using blurring
6. Code for using adaptive Gaussian thresholding

Mittal Madhav

1. Code for perspective transform
2. Code for skew correction
3. Code for speckle noise removal
4. Code for thinning
5. Writing the report

Introduction

Optical character recognition (OCR) aims to recognize texts in imaged documents. It is one of the earliest addressed computer vision tasks, since in some aspects it does not require deep learning. Thus, some of the earliest OCR implementations can be dated back till the early twentieth century.

OCR usually involves a series of image processing and recognition tasks. These tasks include

1. Text image binarization that converts a colour/grayscale image into a binary image with multiple foreground regions (usually characters)
2. Connected component labelling that detects each binarized character region
3. Character recognition by using some classifiers such as a pre-trained neural network

In this project, we aim to implement the first step of OCR, i.e., text image binarization. We implement the Otsu Global Thresholding Algorithm to binarize the image, and then modify it in order to achieve better and more accurate results.

Different preprocessing steps are required for different images in order to ensure robust and accurate character recognition. Since the images usually contain some form of noise, it becomes necessary for us to pre-process the input images. These preprocessing steps include

1. Perspective transform
2. Skew correction
3. Binarization
4. Noise removal
5. Thinning

Methods

a. Calculating OCR accuracy

In order to check and compare the efficiency of the algorithms developed in this project, a standard accuracy measurement scale was required to be developed. Thus, we developed an algorithm to check the accuracy of OCR using cosine similarity.

Firstly, two lists of words were made, one consisting of the original text that needed to be manually typed in, and the other consisting of the text detected by the algorithm. These words were then tokenized (broken down into syllables) using the word_tokenize() function from Python NLTK library.

```
# Tokenization
original_tokens = word_tokenize(original)
ocr_tokens = word_tokenize(ocr)
```

Fig. 1 Tokenization of words

A list of stop words were then extracted from these token lists. These stop words and tokens are used for calculation of cosine similarity between the original text and the text detected by the algorithm.

```
# Get cosine similarity
for i in range(len(union)):
        c += original_list[i] * ocr_list[i]
cosine = c / float((sum(original_list)*sum(ocr_list))**0.5)
```

Fig. 2 Calculating Cosine Similarity

b. Plotting and showing the pixel-intensity histogram of images

It is necessary to plot the pixel-intensity histogram of images in order to get a better visual idea about the contrast and the general spread of pixel values of the image. An image with a bi-modal histogram would be much easier to implement binarization on as compared to any other histogram due to a clear point of separation.

First, a get_histogram() function is defined in order to obtain the pixel intensity values of the provided image in the form of an array, as well as the bin edges.

```
# To calculate and plot histogram of image
def get_histogram(img):
  hist, bins = np.histogram(img, 256, [0, 256])
  return hist, bins
```

Fig. 3 : Function to return intensity values

A separate function is then used to display the corresponding histogram, using the values calculated from this function.

```
def show_histogram(hist, bins):
  width = 0.7 * (bins[1] - bins[0])
  center = (bins[:-1] + bins[1:]) / 2
  plt.bar(center, hist, align='center', width=width)
  plt.show()
```

Fig. 4 : Function to plot the pixel-intensity histogram of an image

An input image and its corresponding histogram can be seen below

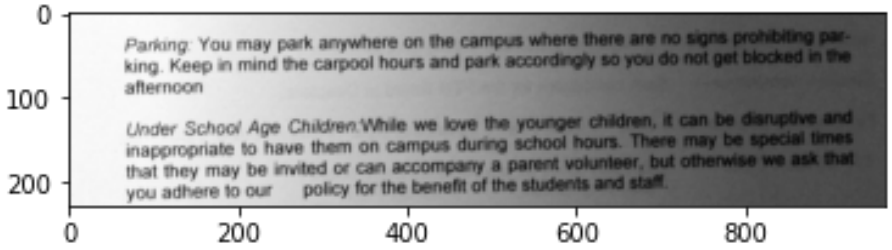


Fig. 5 : Original Image

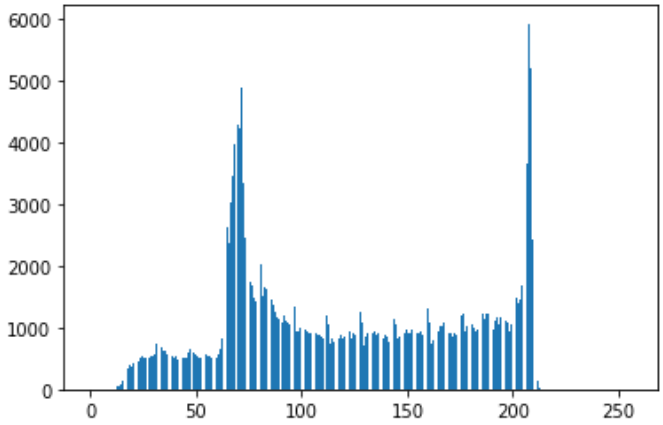


Fig. 6 : Pixel-intensity histogram

Experiments and Results

1. Question 1: Otsu thresholding

a. OCR without Otsu

Description

The two sample images were first tested using the PyTesseract OCR algorithm without binarization. The sample images are shown in the figures below.

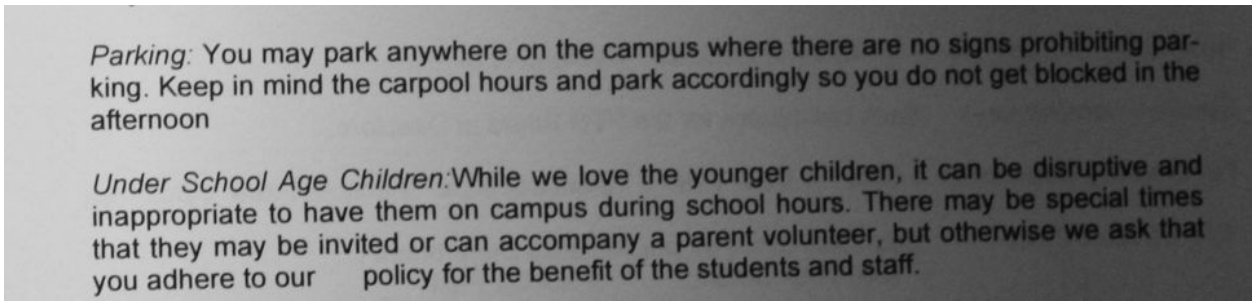


Fig. 7 Sample Image 1

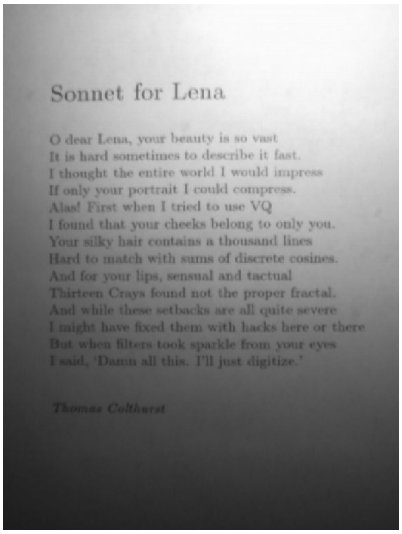


Fig. 8 Sample Image 2

Output

```
First image:
Parking: You may park anywhere on the ce
king. Keep in mind the carpool hours and park
afternoon

Under School Age Children:While we love
inappropriate to have them on campus @ )
that they may be invited or can accompany :
you adhere to our _ policy for the benefit of
```

Fig. 9 : PyTesseract Output for Sample Image 1

```
Second image:
Sonnet for Lena
```

Fig. 10 : PyTesseract Output for Sample Image 2

Results

We find out that the OCR Accuracy without Otsu Thresholding for Sample Image 1 is 0.699 (approx), and for Sample Image 2 is 0.170 (approx). The low accuracy for Sample Image 1 can be attributed to the “Shadow” noise which leads to the PyTesseract OCR algorithm to cause mislabelling. Similarly, Sample Image 2 is contaminated with “Shadow” noise. Furthermore, due to its low resolution, the PyTesseract OCR algorithm fails to work, giving it an accuracy of just 0.170.

These results show the need of a good binarization algorithm in order to separate the text from the background, so as to increase the efficiency of the OCR algorithm.

b. OCR with Otsu

Description

In image processing and analysis, we sometimes need a method to separate two related data, for example, background and foreground, text and paper, etc. Otsu Global Thresholding is one such method which can find the optimal threshold to distinguish two-class data. This method can thus be applied in image segmentation and image binarization tasks.

For example, Fig. 7 shows a grayscale image of a leaf that needs to be binarized and its corresponding histogram. Otsu's algorithm would find an optimal threshold value, such that any pixel values above that threshold would be classified as the leaf, and any value below that would be the background. The red line signifies the computed Otsu threshold value.

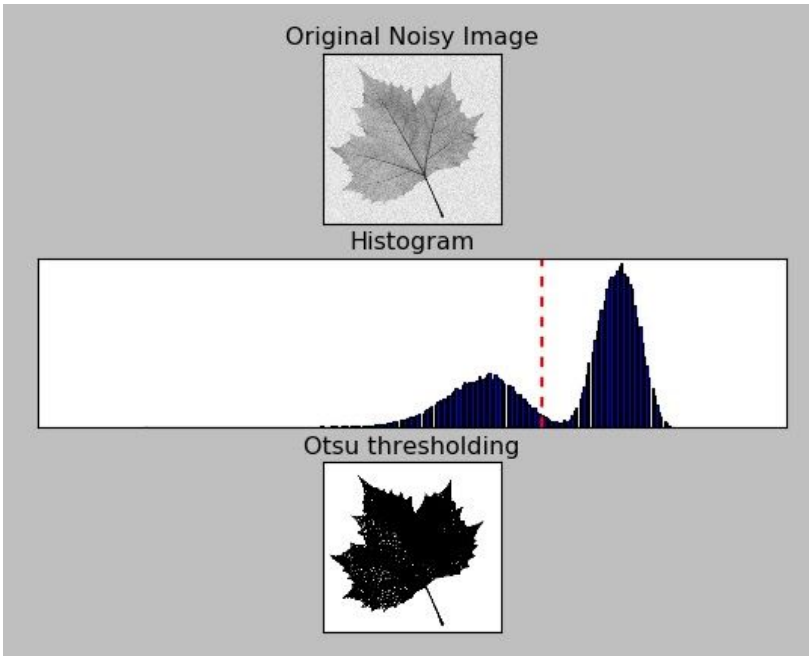


Fig. 7 Implementation Example of Otsu Threshold Algorithm[1]

In this project, Otsu's method is used to reduce a grayscale image to a binary image. It assumes that the image contains two classes of pixels following bi-modal histogram (foreground pixels and background pixels) and generates a threshold based on the histogram of an image.

[1] "OpenCV & Python – The Otsu's Binarization for thresholding ...." 13 Apr. 2017, https://www.meccanismocomplesso.org/en/opencv-python-the-otsus-binarization-for-thresholding/. Accessed 25 Nov. 2020.

This ideal Otsu threshold value is calculated by minimizing the weighted within-class variance. This variance is represented by $\sigma^2(t)$ and is given by the following formula.

$$\sigma^2(t) = q_1(t)\,\sigma_1^2(t) + q_2(t)\,\sigma_2^2(t)$$

Fig. 8 Formula for within-class weighted variance

Where,

t is the selected threshold value

$q_1(t)$ is the probability of any given pixel being in class 1

$q_2(t)$ is the probability of any given pixel being in class 2

$\sigma_1^2(t)$ is the variance of class 1

$\sigma_2^2(t)$ is the variance of class 2

Thus,

$q_1(t)\,\sigma_1^2(t)$ represent the weighted variance of class 1

$q_2(t)\,\sigma_2^2(t)$ represent the weighted variance of class 2

Since we need to obtain the threshold value such that $\sigma^2(t)$ is minimized, we iterate over all the possible threshold values (1 - 255) and obtain the weighted variance for each of them, and choose the threshold value with the least weighted variance. The implementation can be seen in the code snippet below.

```
# To get threshold value which minimzes intra class variance
def get_optimal_threshold(histogram):
  pixel_count = get_pixel_count(histogram)
  optimal_threshold = 0
  lowest_variance = -1
  for threshold in range(1, 256):
    try:
      weighted_variance_L = get_weighted_variance(histogram, 0, threshold, pixel_count)
      weighted_variance_R = get_weighted_variance(histogram, threshold+1, 256, pixel_count)
      weighted_variance_total = weighted_variance_L + weighted_variance_R
    except:
      continue

    # First iteration or lower variance value
    if lowest_variance == -1 or weighted_variance_total < lowest_variance:
      lowest_variance = weighted_variance_total
      optimal_threshold = threshold

  return optimal_threshold
```

Fig. 9 Implementation of Otsu's within class weighted variance

Output

The two sample images were first binarized using Otsu Global Thresholding and then the PyTesseract OCR algorithm was used on these thresholded images. The output images can be seen in the figures below.

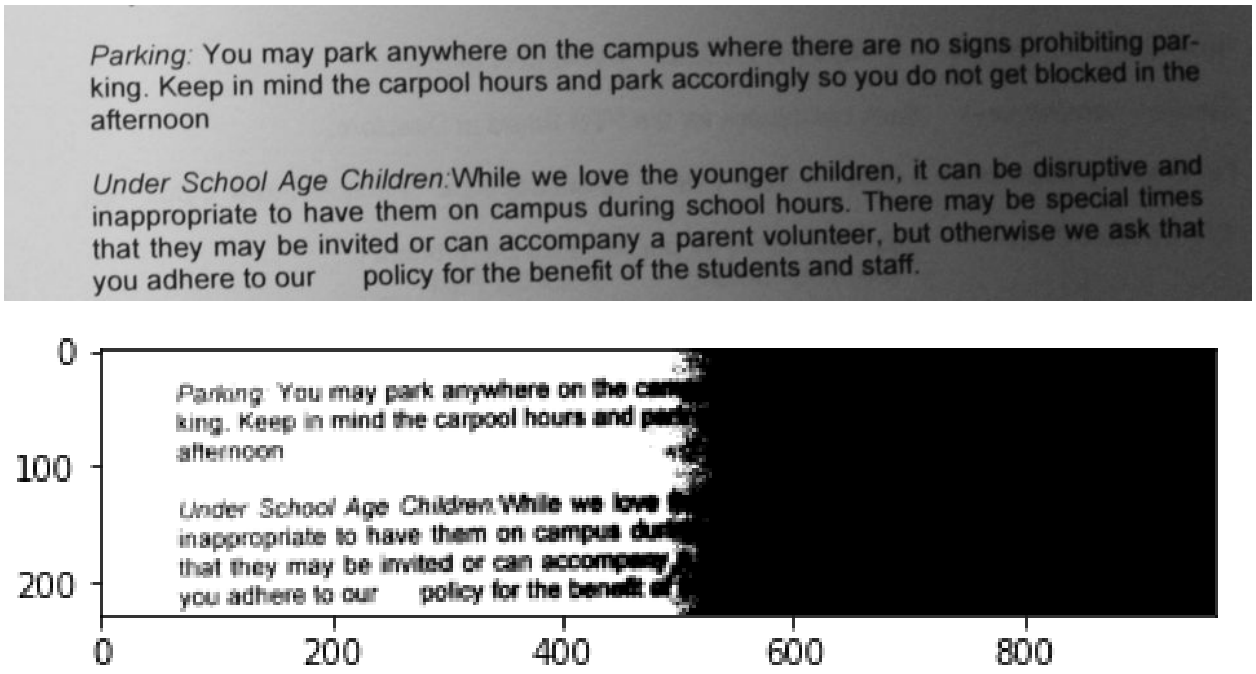


Fig. 10 Sample image 1 before and after global thresholding

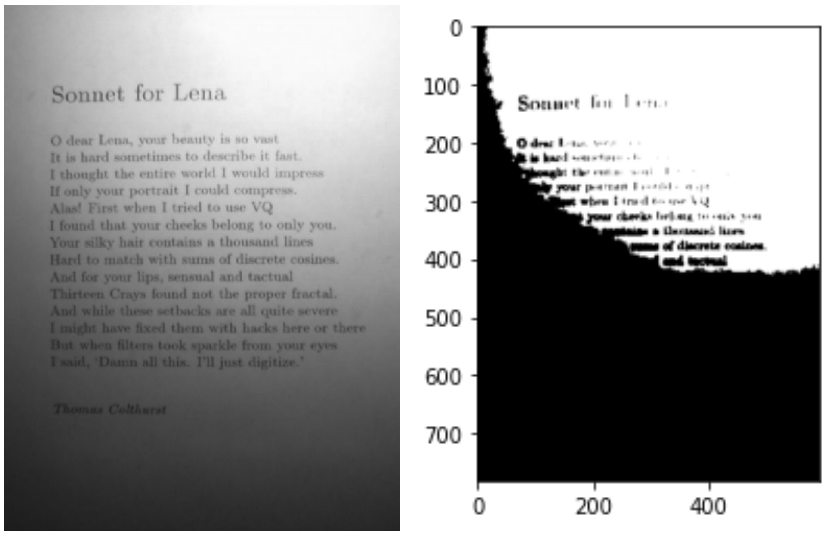


Fig. 11 Sample image 2 before and after global thresholding

Results

The OCR accuracies of the sample images can be found in Table 1.

S.No	Accuracy without Otsu Thresholding	Accuracy with Global Otsu Thresholding
Sample Image 1	0.699	0.699
Sample Image 2	0.170	0.000

Tab. 1 Comparison of OCR accuracies with and without Otsu Global Thresholding

As we can see from the table, after global thresholding, the OCR accuracies remains the same for the first image, and drops to 0 for the second image. This behaviour can be seen in many other images as well. This is so because in most of the cases, the image to be binarized is not ideal, and may be contaminated with noise.

For example, in this project, we work on two images and both of them are contaminated with shadows. In the conventional thresholding method, a global or standard threshold value is taken as the threshold for all the pixels in the image. While this standard threshold value might be a good approximation for a global average, it may not be able to work in certain sections with noise.

In the case of “Shadow” noise, the background region, which is supposed to be below the threshold value in order to be binarized to a value of 255 (white), might actually end up having a darker area and thus might be misclassified as text pixels. This ends up misclassifying the entire shadow region as text and thus turns all the pixels to a value of 0 (black). This is why Figures 10 and 11 have big areas of all dark pixels. This leads to a low classification accuracy.

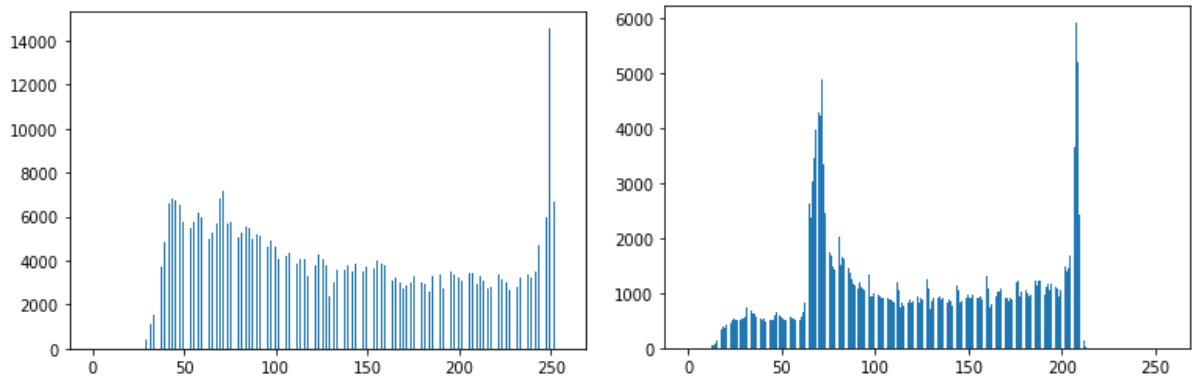


Fig. 12 Pixel Intensity Histograms for sample images 1 (left) and 2 (right)

As it can clearly be seen from the histograms, due to the shadow noise, the histogram is flattened and thus has no clear threshold value (since it is not a bimodal distribution anymore). This leads to misclassification of pixels, and adversely affects the OCR accuracy.

2. Question 2: Improving thresholding

a. Adaptive thresholding

Description

In image processing, thresholding is applied to obtain binary images from grayscale images. For the reasons mentioned in the previous section, global thresholding is not appropriate for images with noise, and thus needs modification. Thus, we use adaptive thresholding.

This shortcoming of Otsu's Global Thresholding can be overcome by adaptive thresholding. In this method, the image is broken up into smaller blocks and Otsu Thresholding is conducted individually in each of these blocks, which allows us to learn region-specific thresholds. This works because in an area having the "Shadow" noise, the weighted variance will be calculated according to that entire region with shadow noise, resulting in a cleaner, more accurate image.

```
def adaptive_otsu(img, row_block_size, col_block_size):
  assert col_block_size <= img.shape[1] and row_block_size <= img.shape[0]

  output = np.zeros_like(img)

  # Split the image into blocks and run Otsu
  for row in range(0, img.shape[0], row_block_size):
    for col in range(0, img.shape[1], col_block_size):
      row_start, row_end = row, min(img.shape[0], row+row_block_size)
      col_start, col_end = col, min(img.shape[1], col+col_block_size)

      block = img[row_start:row_end, col_start:col_end]
      otsu_block = otsu(block)
      output[row_start:row_end, col_start:col_end] = otsu_block

  return output
```

Fig. 13 Implementation of Adaptive Otsu Thresholding

Outputs

The two sample images were first binarized using Adaptive Otsu Thresholding and then the PyTesseract OCR algorithm was used on these thresholded images. The output images along with the recognized text can be seen in the figures below.

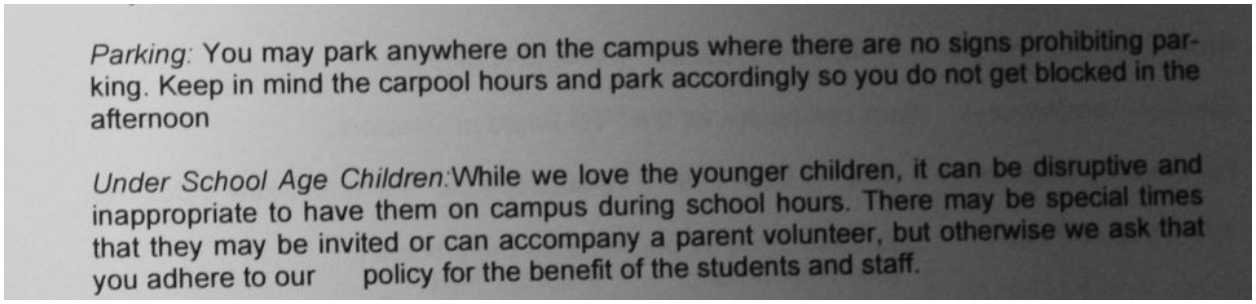
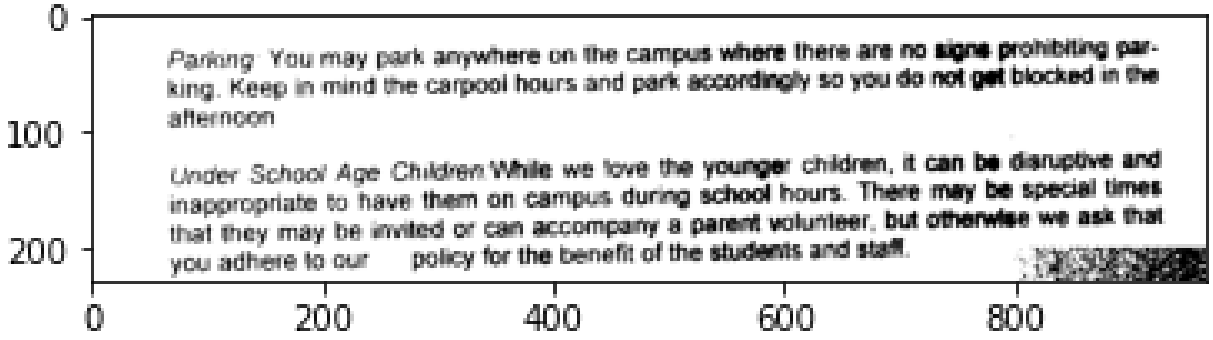


Fig. 14 Sample image 1 before and after adaptive thresholding



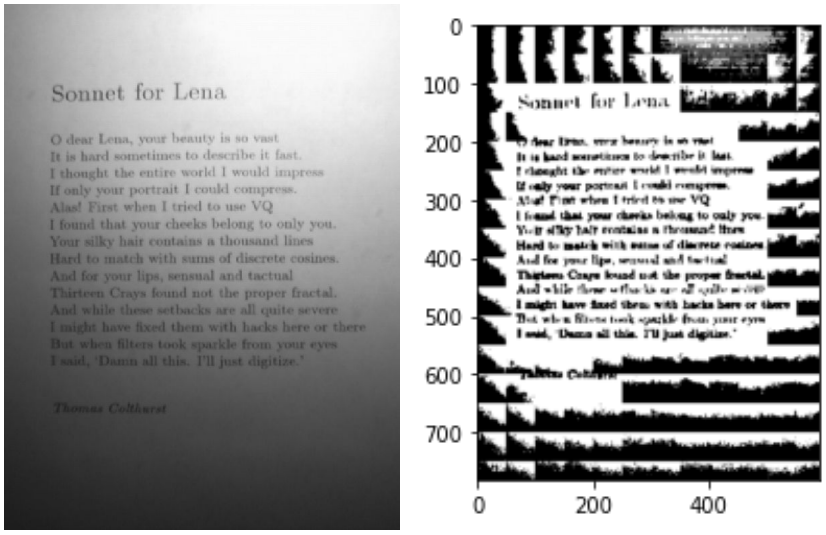


Fig. 15 Sample image 2 before and after adaptive thresholding

Results

The OCR accuracies of the sample images can be found in Table 2.

S.No	Accuracy without Otsu Thresholding	Accuracy with Adaptive Otsu Thresholding
Sample Image 1	0.699	0.936
Sample Image 2	0.170	0.276

Tab. 2 Comparison of OCR accuracies with and without Otsu Adaptive Thresholding

As it can be seen from the table, adaptive otsu thresholding dramatically increases the OCR accuracies for both the images. This is because it takes in consideration the noise, by taking different region-specific thresholds.

Adaptive thresholding, however, is not perfect. It does not work well in large spaces without text, due to the absence of a bimodal distribution. In essence, adaptive otsu fails in cases where there are large blank spaces with changing intensity of shadow noise, similar to global thresholding.

This is evident from Fig. 12. In Sample Image 1, there are no such wide blank spaces, which is why adaptive thresholding works well on it. However, Sample Image 2 has many wide empty spaces, leading to incorrect classification.

b. Background removal using variance thresholding

Description

As we saw in the previous section, adaptive thresholding does not work perfectly in some cases of “Shadow” noise. This happens because the window size may not be selected properly, and so within the individual blocks, only a percentage of the pixels may be contaminated with noise. This results in the same problem as was described in the previous section with global thresholding.

This can be solved by background removal. In a text image, we only need to classify the text from the background. Any large space with similar pixels can be classified as the background, since just text cannot cover a large space, it needs to have some gaps.

Thus, we introduce a novel background removal algorithm which works on the basis of variance thresholding. This algorithm works in unison with adaptive thresholding. In each iteration of adaptive thresholding, we check the histogram of the current block. If the block has some text, then it is bound to have a high variance distribution , because the text is black and the background will be lighter than the text (regardless of “shadow” noise). If the block does not have any text, and thus only has the background, then it is bound to have a unimodal, low variance distribution (regardless of “shadow” noise, because in a small block the variance between shadow blocks is usually very low).

The histograms and corresponding variances of different types of blocks can be seen in the figure below.

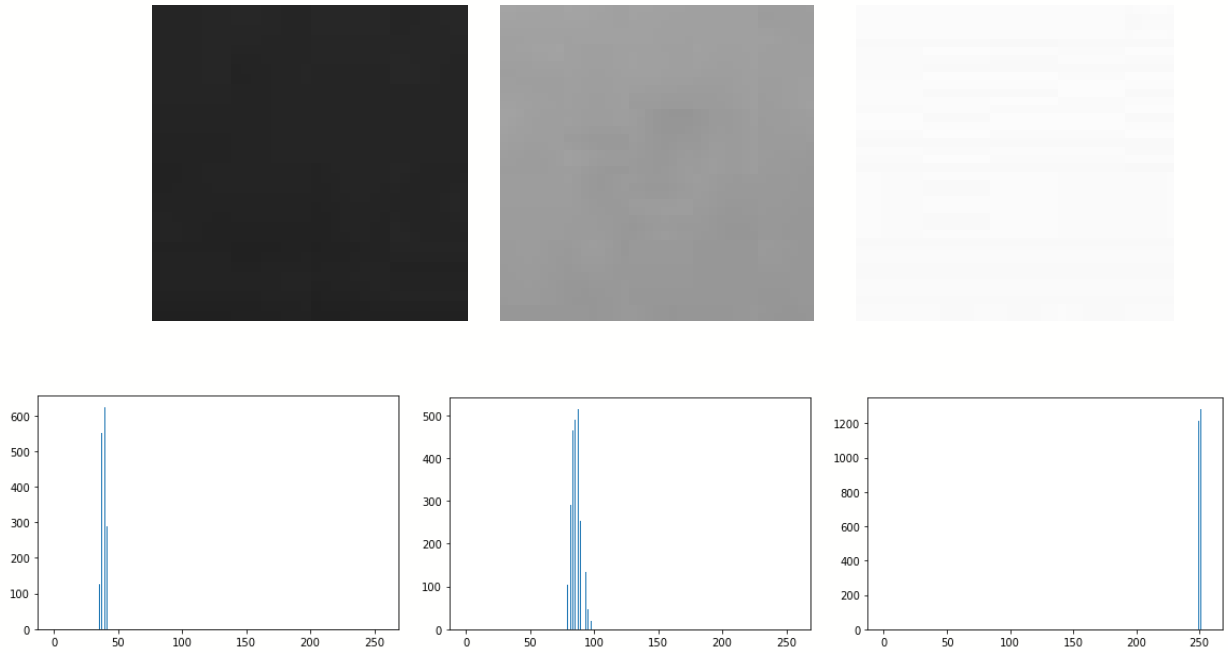


Fig. 16 Background blocks and their corresponding histograms

These histograms are for background blocks that don’t have text. They are black, grey and white in color with variances 2.74, 20.52 and 0.93 respectively. These are all relatively low variances, which occur due to the absence of text.

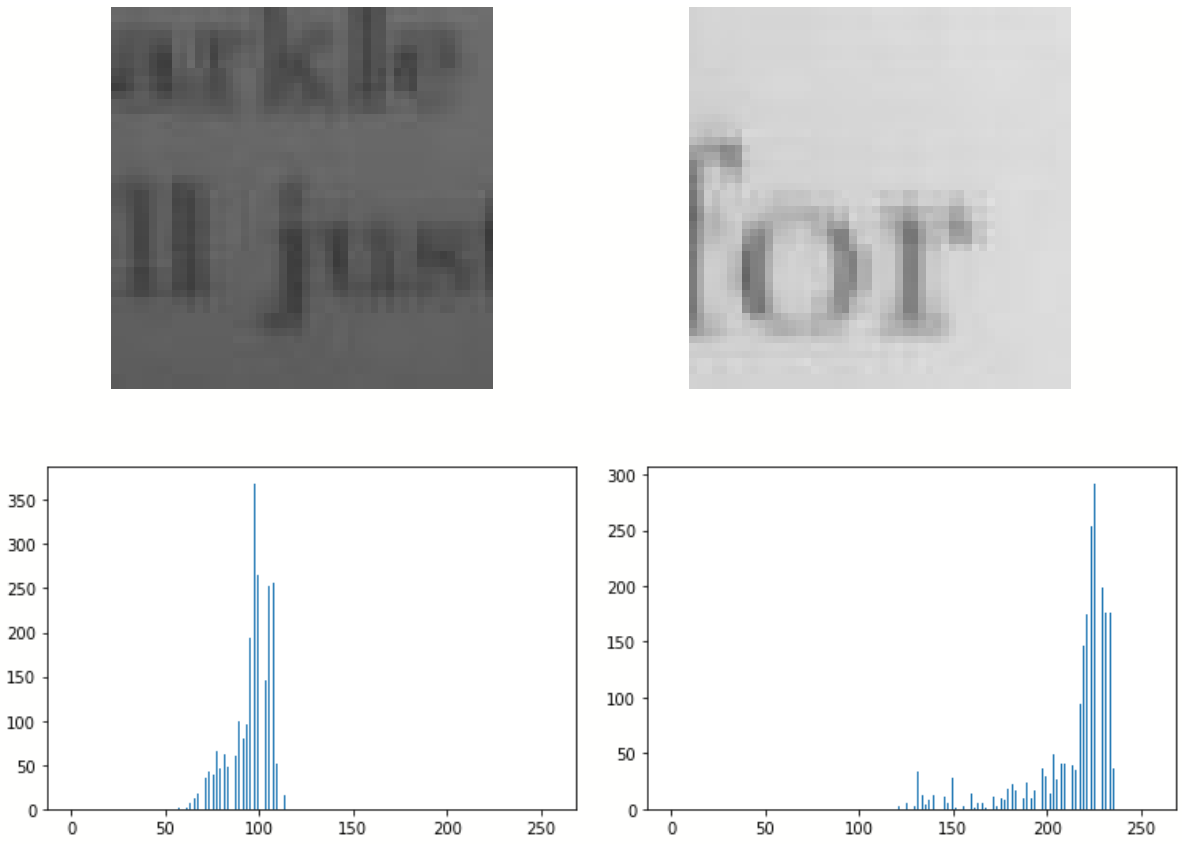


Fig. 17 Text blocks and their corresponding histograms

These histograms are for background blocks that do have text. They are black and white in color with variances 111.97, 479.93 respectively. These are all relatively high variances, which occur due to the presence of text.

We thus create an algorithm to check the variance of each block after each iteration in adaptive thresholding, and if the variance is above a selected threshold variance, then that means there is text in the region, and it is left alone. If the variance is lower than a selected threshold variance, then the region does not have any text, and is thus just background. In this case, the region is turned completely white. In this experiment, the threshold variance was chosen to be 50.

It must be noted that the threshold variance value must be selected by trial and error, and some intuition. Implementation of the background removal algorithm is shown in the code snippet below.

```python
def adaptive_otsu_with_bg(img, row_block_size, col_block_size, detect_bg=False, bg_thresh=50):
  assert col_block_size <= img.shape[1] and row_block_size <= img.shape[0]

  output = np.zeros_like(img)

  # Split the image into blocks and run Otsu
  for row in range(0, img.shape[0], row_block_size):
    for col in range(0, img.shape[1], col_block_size):
      row_start, row_end = row, min(img.shape[0], row+row_block_size)
      col_start, col_end = col, min(img.shape[1], col+col_block_size)

      block = img[row_start:row_end, col_start:col_end]
      otsu_block = otsu(block)

      if detect_bg:
        var = np.var(block)
        cv2_imshow(block)
        print(f"Variance: {var}")
        if var < bg_thresh:
          otsu_block.fill(255) # If variance is low, treat as background

      output[row_start:row_end, col_start:col_end] = otsu_block

  return output
```

Fig. 18 Implementation of Adaptive Thresholding with Background Removal

Output

Variance thresholding was only applied on Sample Image 2, since only this image suffered from degradation due to noise after adaptive thresholding. Sample image 2 was first binarized using Variance Thresholding and then the PyTesseract OCR algorithm was used on it. The input and output image can be seen below.

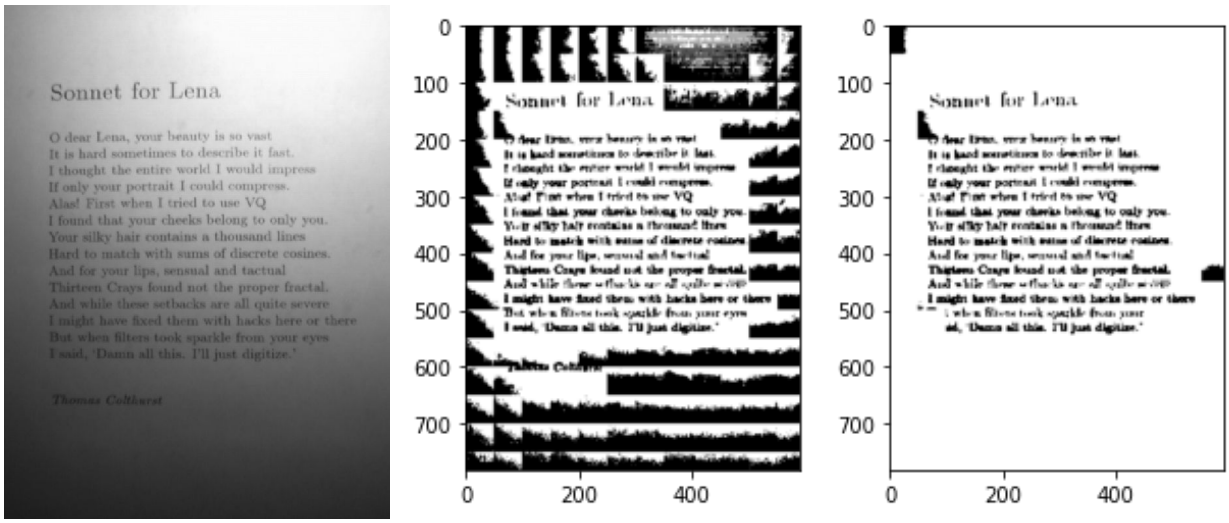


Fig. 19 Sample image 2 originally vs. before and after variance thresholding

Results

The OCR accuracies of Sample Image 2 can be found in Table 3.

S.No	Accuracy without Otsu Thresholding	Accuracy with Adaptive Thresholding	Accuracy with Variance Thresholding
Sample Image 2	0.170	0.276	0.315

Tab. 3 Comparison of OCR accuracies originally vs. before and after variance thresholding

As it can be seen from the table, variance thresholding also increases the OCR accuracy. This is because it removes the unwanted noise caused by the shadows. It can also be seen from Fig. 16 that most of the background pixels that were incorrectly categorized as text pixels were corrected. Thus, variance thresholding works sufficiently well.

c. Background removal using blurring

Description

Another method to address the problem of “Shadow” noise is gaussian blurring. This method takes advantage of a specific property of the gaussian filter, i.e., gaussian smoothing. The Gaussian kernel is often used for gaussian smoothing, or blurring images to remove details and noise. As we increase the size of the gaussian kernel, more and more noise is removed, but the image also gets blurrier.

By exploiting this property of the gaussian filter, we can remove the “Shadow” noise. If we increase the size of the kernel to a large enough value, it will remove a large amount of details, and successfully blur the image. When applied to the text images with shadow noise, it will remove the details (text) and blur the background (shadows).

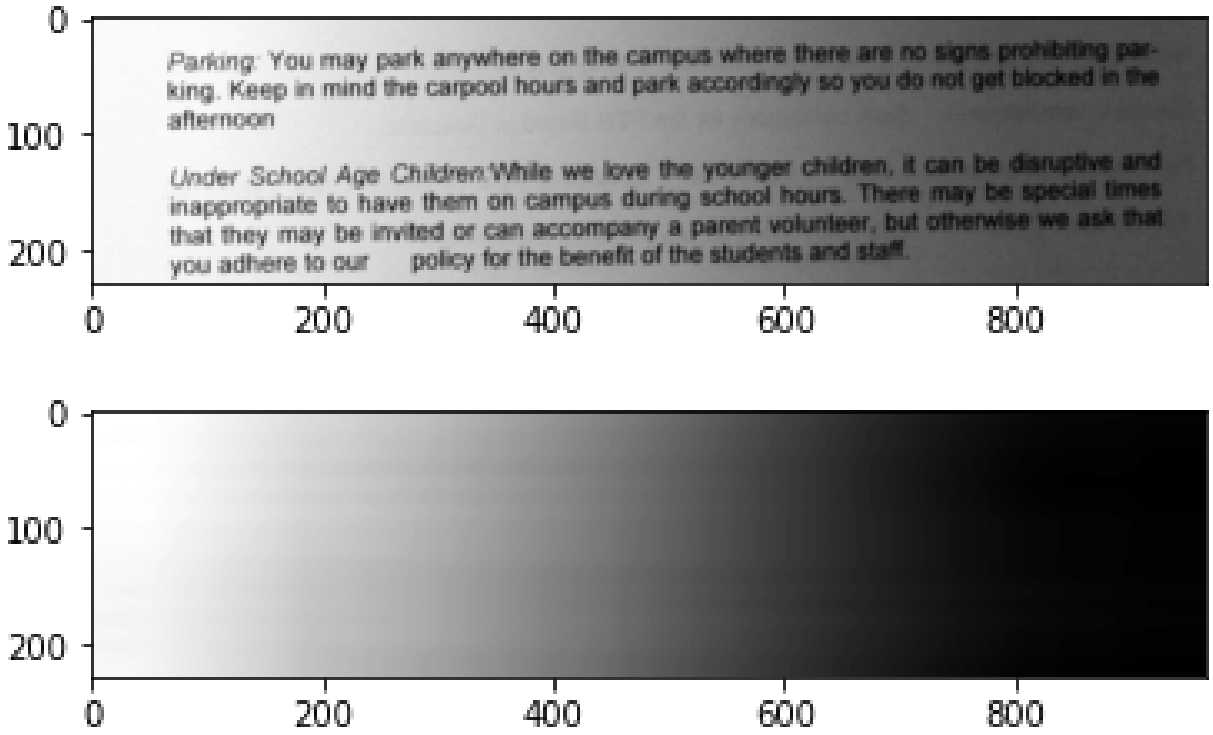


Fig. 20 Sample Image 1 before and after gaussian smoothing

After acquiring the background with shadows, we can subtract the background from the original image to obtain just the text image. This image is then sent through the adaptive thresholding algorithm. The implementation can be seen in the code snippet below.

```
def remove_bg(img, kernel_size=5):
  blurred_img = cv2.GaussianBlur(img,(kernel_size,kernel_size),0)
  no_bg_img = blurred_img - img
  return no_bg_img
```

Fig. 21 Code implementation of Gaussian Smoothing

Output

The output images for the sample images can be seen in the figures below.

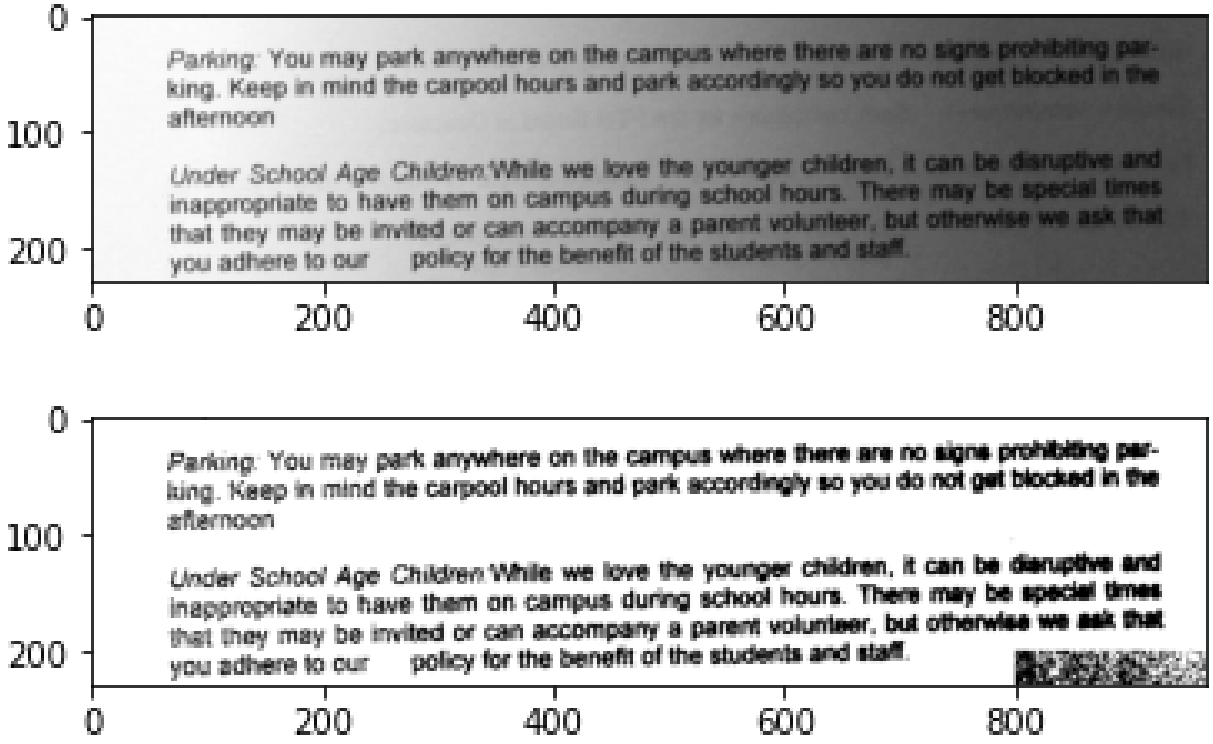


Fig. 22 Sample Image 1 before and after background subtraction and adaptive thresholding

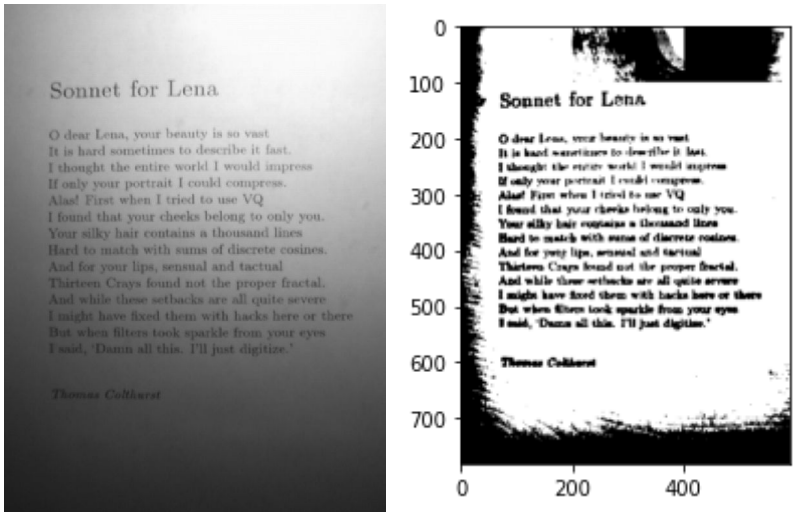


Fig. 23 Sample Image 2 before and after background subtraction and adaptive thresholding

Results

The OCR accuracies of Sample Image 1 and 2 can be found in Table 4.

S.No	Accuracy without Otsu Thresholding	Accuracy with Adaptive Thresholding	Accuracy with Variance Thresholding	Accuracy with Gaussian Blurring
Sample Image 1	0.699	0.936	-	0.875
Sample Image 2	0.170	0.276	0.315	0.444

Tab. 4 Comparison of OCR accuracies originally vs. before and after variance thresholding

As it can be seen from the table, gaussian blurring increases the OCR accuracy for Sample Image 2, but decreases the accuracy for Sample Image 1. This may be because shadow in Sample Image 1 is very gradual, so the gaussian blurring causes mixing up of dark and light regions. In Sample Image 2, the shadow is abrupt and thus there is no mixing of dark and light pixels. Thus, gaussian blurring works well on this image.

d. Adaptive Gaussian Thresholding

Description

All the architectures till now consisted of Adaptive Otsu Thresholding in order to calculate multiple thresholds across an image. While this is a good way of preventing generalization of threshold value and avoiding shadow noise, there are still better methods for binarization of images.

One such method is Adaptive Gaussian Thresholding. Both Otsu and Gaussian Thresholding are adaptive algorithms, thus both of these algorithms calculate different

thresholds for smaller regions. The difference, however, is in how these individual thresholds are calculated.

Otsu Thresholding emphasizes on choosing the threshold that minimises the weighted within-class variance. The adaptive gaussian thresholding algorithm takes a gaussian-weighted sum of the neighbourhood values minus the given constant **C**. This is better than adaptive Otsu since it weighs the values of closer pixels more and guarantees a highly localised threshold.

We implement the Adaptive Gaussian Thresholding algorithm using the openCV library as shown in the code snippet below.

```
def adaptive_gaussian(img, block_size, C):
  thresholded = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, block_size, C)
  return thresholded
```

Fig. 24 Code implementation of Adaptive Gaussian Thresholding

Output

The output images for the sample images can be seen in the figures below.

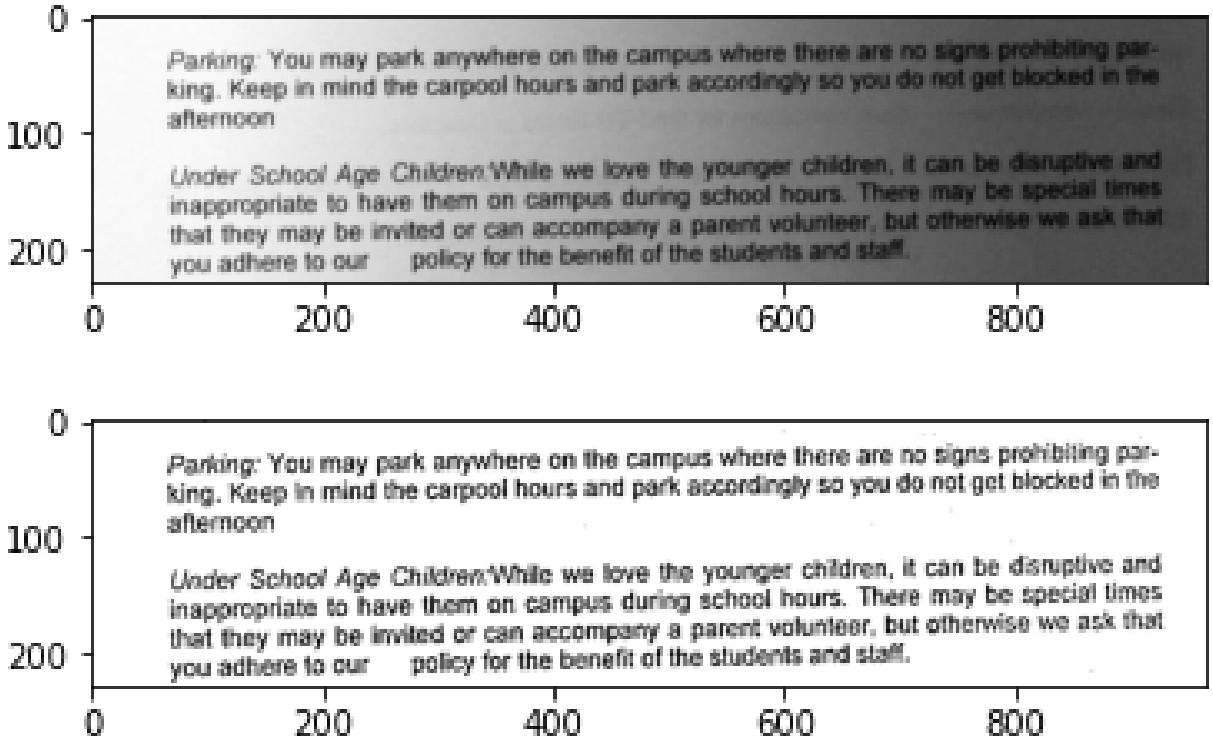


Fig. 25 Sample Image 1 before and after Adaptive Gaussian Thresholding

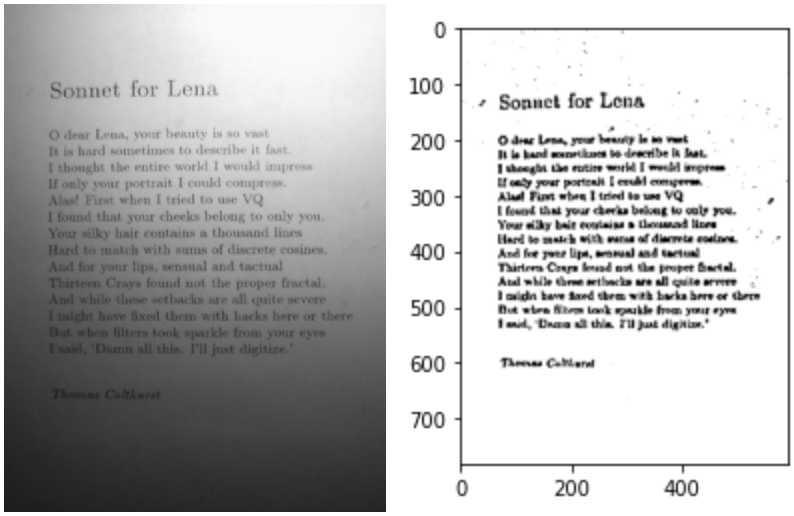


Fig. 26 Sample Image 2 before and after Adaptive Gaussian Thresholding

Results

The OCR accuracies of Sample Image 1 and 2 can be found in Table 5.

S.No	Accuracy without Otsu Thresholding	Accuracy with Adaptive Otsu Thresholding	Accuracy with Variance Thresholding	Accuracy with Gaussian Blurring	Accuracy with Adaptive Gaussian Thresholding
Sample Image 1	0.699	0.936	-	0.875	1.0
Sample Image 2	0.170	0.276	0.315	0.444	0.649

Tab. 5 Comparison of OCR accuracies originally vs. before and after Adaptive Gaussian Thresholding

As it can be seen from the table, Adaptive Gaussian Thresholding dramatically increases the OCR accuracy. It is a really efficient algorithm since it weighs the values of closer pixels more and guarantees a highly localised threshold.

3. Question 3: More robust character recognition

The previous section emphasised on improving the OCR accuracy for Sample Images 1 and 2, and since they suffered from “Shadow” noise, thus we only focused on removing that particular kind of noise.

However, in the real world, the images may suffer from several different types of image degradation. Even with a really good OCR algorithm, we may get a bad OCR accuracy just because of the image degradation. To take care of this, image pre-processing must be done.

These pre-processing steps include

a. Perspective Transform
b. Skew Correction
c. Binarization
d. Noise Removal
e. Thinning

It must be noted that we use the Adaptive Otsu Thresholding for binarization of images throughout this part of the report, since it is what the project required us to implement.

a. Perspective Transform

A common problem is that most images are taken at an angle from the actual plane of the paper. This means that the text is not horizontal in the image which can sometimes confuse the OCR algorithm. We will apply automatic perspective transformation to ensure that most images are transformed from a 3D plane to a 2D plane.

An example image which needs perspective transform, along with its binarized image is shown in the figure below.

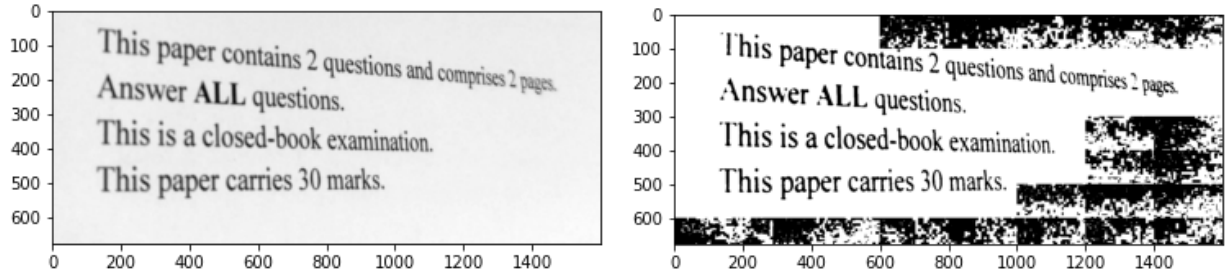


Fig. 27 Image before and after adaptive thresholding

The skew in the image can be clearly seen in Fig. 24. This skew affects the binarization algorithm's accuracy, as well the PyTesseract's OCR algorithm accuracy, giving a final accuracy of 0.671.

Thus, we build a function for transforming the perspective of the image. The functions inputs the image and locations for the four text corners, and outputs the perspective transformed image. The code implementation of this function can be seen in the figure below.

```python
def perspective_transform(img, corners):
  (tl, tr, br, bl) = corners

  # Compute width of new image
  width_top = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2)) # Distance between top right and left
  width_bottom = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2)) # Distance between bottom right and left
  width = max(int(width_top), int(width_bottom)) # Maximum of above 2

  # Compute height of new image
  height_left = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2)) # Distance between left top and bottom
  height_right = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2)) # Distance between right top and bottom
  height = max(int(height_right), int(height_left))

  # Creating points to map corner points to in new image
  corners_new = np.array([
    [0, 0],
    [width - 1, 0],
    [width - 1, height - 1],
    [0, height - 1]], dtype = "float32")

  # Compute perspective transform matrix and transform image
  transform_matrix = cv2.getPerspectiveTransform(corners, corners_new)
  transformed_image = cv2.warpPerspective(img, transform_matrix, (width, height))

  return transformed_image
```

Fig. 28 Code implementation of perspective transform

The image is then passed through this function for pre-processing, followed by binarization. The results can be seen below.

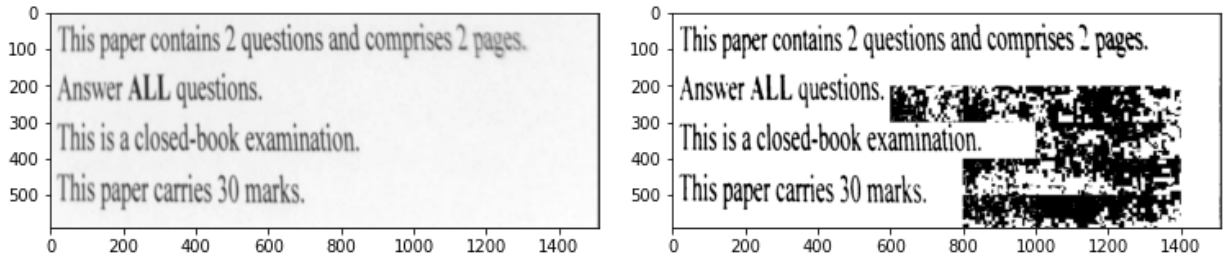


Fig. 29 Perspective transformed image before and adaptive thresholding

As it can be seen from Fig. 26, the function successfully transforms the perspective of the input image. This makes it easier for the PyTesseract OCR algorithm to recognize text, improving the accuracy to 0.888.

b. Image De-skewing

Image obtained from the previous stage may not be correctly oriented, as it may be aligned at any angle. So we need to perform skew correction to make sure that the image forwarded to subsequent stages is correctly oriented.

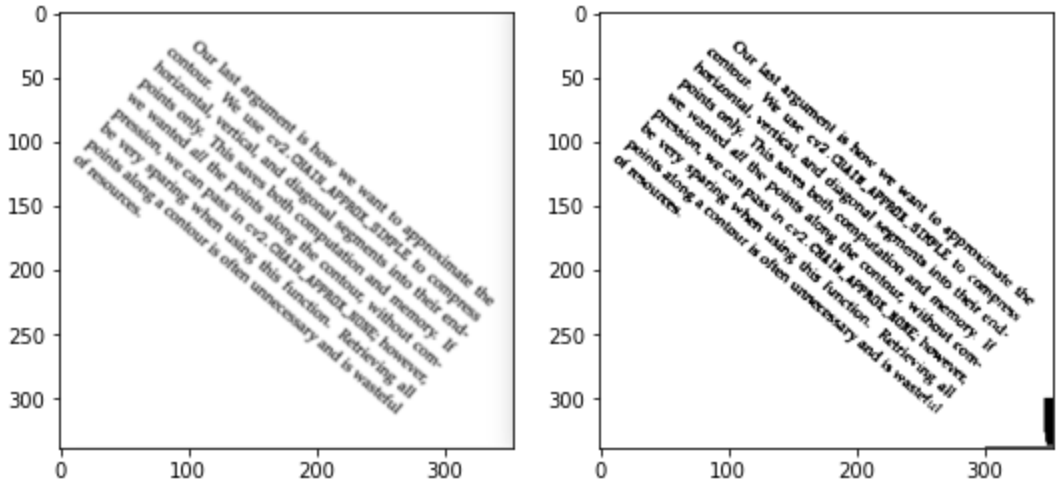


Fig. 30 Image before and after adaptive thresholding

The skew in the image can be clearly seen in Fig. 27. This skew drastically affects PyTesseract's OCR accuracy, dropping it to 0%. This may be because of PyTesseract's inability to parse angle text. Thus, images with high skew angles must be pre-processed.

We build functions for correcting the skew of the image. The functions input the image and outputs the de-skewed image. The code implementation of these functions can be seen in Fig. 28.

First, the skew angle is calculated. This involves a lot of pre-processing steps, including blurring, thresholding and dilating in order to separate out different blocks of text, followed by contour detection and finally calculating the skew angle.

```python
def getSkewAngle(cvImage):

    # Prep image, copy, convert to gray scale, blur, and threshold
    newImage = cvImage.copy()
    blur = cv2.GaussianBlur(newImage, (9, 9), 0)
    thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)[1]

    # Apply dilate to merge text into meaningful lines/paragraphs.
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (30, 5))
    dilate = cv2.dilate(thresh, kernel, iterations=5)

    # Find all contours
    contours, hierarchy = cv2.findContours(dilate, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    contours = sorted(contours, key = cv2.contourArea, reverse = True)

    # Find largest contour and surround in min area box
    largestContour = contours[0]
    minAreaRect = cv2.minAreaRect(largestContour)

    # Determine the angle. Convert it to the value that was originally used to obtain skewed image
    angle = minAreaRect[-1]
    if angle < -45:
        angle = 90 + angle
    return -1.0 * angle
```

Fig. 31 Code implementation for finding skew angle

This skew angle is then used by another function to rotate (de-skew) the image.

```python
# Rotate the image around its center
def rotateImage(cvImage, angle: float):
    newImage = cvImage.copy()
    (h, w) = newImage.shape[:2]
    center = (w // 2, h // 2)
    M = cv2.getRotationMatrix2D(center, angle, 1.0)
    newImage = cv2.warpAffine(newImage, M, (w, h), flags=cv2.INTER_CUBIC, borderMode=cv2.BORDER_REPLICATE)
    return newImage
```

Fig. 32 Code implementation for rotating the image

The image is then passed through this function for pre-processing, followed by binarization. The results can be seen below.

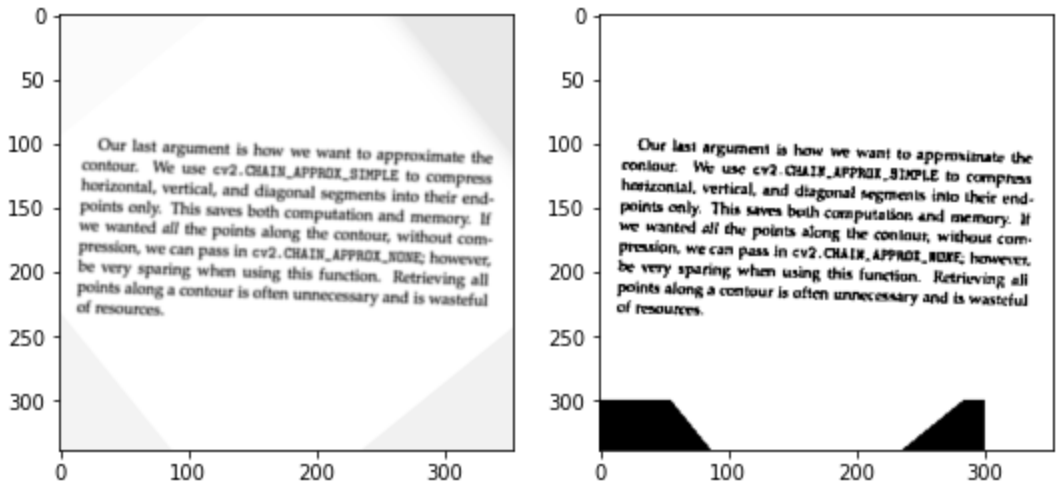


Fig. 33 De-skewed image before and adaptive thresholding

As it can be seen from Fig. 26, the function successfully de-skews the input image. This makes it easier for the PyTesseract OCR algorithm to recognize text, improving the accuracy to 0.637.

c. Removing Speckle Noise

Speckle noise (small dots or foreground components) may be introduced easily into an image while scanning it during Image Acquisition because of various reasons like low clarity camera, a shadow on image etc. This noise should be removed so that the image will be clean and uniform.

An example image which is contaminated with speckle noise along with its binarized image is shown in the figure below.

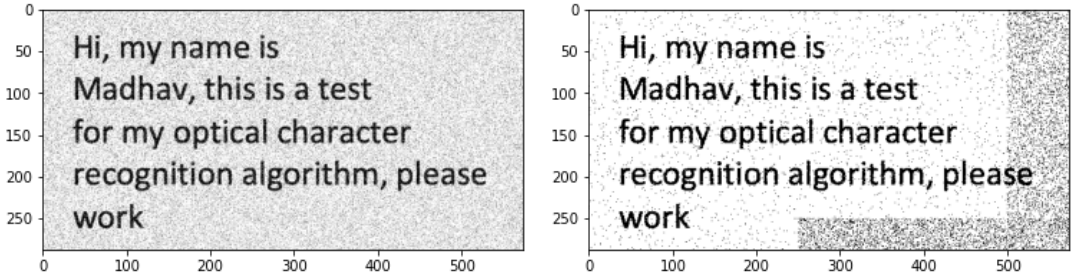


Fig. 34 Image before and after adaptive thresholding

The speckles in the image can be clearly seen in Fig. 31. These speckles vastly affect the binarization algorithm's classification accuracy, and thus PyTesseract's OCR algorithm accuracy, giving a final accuracy of 0.617. This low accuracy is because the speckles are incorrectly recognized as text by Otsu Thresholding.

We can pre-process the image by applying a gaussian filter. As explained before, the Gaussian kernel is often used for gaussian smoothing, or blurring images to remove details and noise. As we increase the size of the gaussian kernel, more and more noise is removed, but the image also gets blurrier. We choose a gaussian kernel size of 9 x 9.

The image is passed through the gaussian filter and adaptive thresholding. The results can be seen in the figure below.

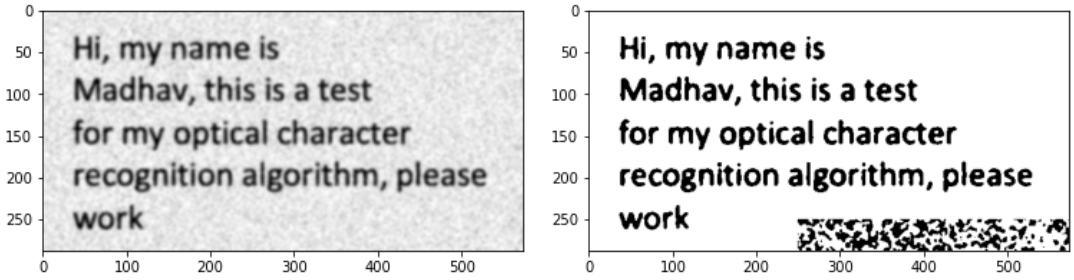


Fig. 35 Denoised image before and after adaptive thresholding

As it can be seen from Fig. 32, the function successfully de-noises the input image. This makes it easier for the PyTesseract OCR algorithm to recognize text, improving the accuracy to 0.880.

d. Thinning

Thinning in OCR is required mostly with handwritten documents, when the text is too thick and thus some characters overlap with the others. Thinning reduces the width of each character, thus reducing the overlap and making it easier for the OCR algorithm to detect individual characters

An example image which requires thinning can be seen in the figure below.

Fig. 36 Image in need of thinning

This image has very thick text which may affect the contours in OCR algorithms, thinning may thus be beneficial for improving the accuracy. This image, thus gives an accuracy of just 0.5 when passed through PyTesseract OCR algorithm. We use the cv2 erode function with a kernel size of 7 x 7 for thinning the image. The code implementation can be seen in Fig. 34.

```
kernel = np.ones((7,7),np.uint8)
thinned_comp1 = cv2.erode(comp1, kernel,iterations = 1)

display(thinned_comp1)
```

Fig. 37 Code implementation of thinning using cv2

FIg. 38 Image after thinning

This thinned image gives an improved accuracy of 1.0, showing that thinning can be used to improve the OCR accuracy for images with thick, overlapping text.

Conclusions

This project aimed at improving the PyTesseract OCR accuracy by implementing different binarization techniques on two sample images. We first looked at the PyTesseract OCR **accuracies when the sample images were not binarized**. The accuracies for sample images 1 and 2 **were found to be 0.699 and 0.170 respectively**. These low accuracies could be attributed to the “Shadow” noise for image 1, and low resolution for image 2.

We then implemented **Global Otsu Thresholding** in order to binarize both the images. We found out that contrary to our understanding, the accuracies actually deteriorated. We found out that the **accuracies were 0.699 and 0**. This happened because while the chosen global threshold value might be a good approximation for a global average, it may not be able to work in certain sections with noise. It resulted in misclassification of entire regions contaminated with “Shadow” noise as text (black pixels), leading to low classification accuracies.

S.No	Accuracy without Otsu Thresholding	Accuracy with Global Otsu Thresholding
Sample Image 1	0.699	0.699
Sample Image 2	0.170	0.000

Tab. 6 Comparison of OCR accuracies with and without Otsu Global Thresholding

The pitfalls of Global Otsu Thresholding were overcome by a custom algorithm - **Adaptive Otsu Thresholding**. In this method, the image was broken up into smaller blocks and Otsu Thresholding is conducted individually in each of these blocks, which allowed us to learn region-specific thresholds. This works because in an area having the “Shadow” noise, the weighted variance will be calculated according to that entire region with shadow noise, resulting in a cleaner, more accurate image. **The OCR accuracies were found to be 0.936 and 0.276**.

S.No	Accuracy without Otsu Thresholding	Accuracy with Adaptive Otsu Thresholding
Sample Image 1	0.699	0.936
Sample Image 2	0.170	0.276

Tab. 7 Comparison of OCR accuracies with and without Otsu Adaptive Thresholding

It was found that adaptive thresholding did not achieve perfect results. It does not work well in large spaces without text, due to the absence of a bimodal distribution. In essence, adaptive otsu fails in cases where there are large blank spaces with changing intensity of shadow noise, similar to global thresholding.

Thus, we created a novel **variance thresholding based method**. It worked in unison with adaptive thresholding, and worked on the principle that background blocks have low pixel intensity variance due to the absence of text, and thus any block below a set threshold variance can be regarded as background block and can be set to 255 pixel value.

Since adaptive thresholding had misclassification errors only on image 2, thus variance thresholding was only used on that image.

S.No	Accuracy without Otsu Thresholding	Accuracy with Adaptive Thresholding	Accuracy with Variance Thresholding
Sample Image 2	0.170	0.276	0.315

Tab. 8 Comparison of OCR accuracies originally vs. before and after variance thresholding

Another method we used to address the problem of “Shadow” noise was **gaussian blurring**. We increased the gaussian kernel to a large size to remove all text and obtain the background, and then removed the background shadows by subtracting the obtained background. It was noticed that while it improved the accuracy for Sample Image 2, it decreased the accuracy for Sample Image 1

S.No	Accuracy without Otsu Thresholding	Accuracy with Adaptive Thresholding	Accuracy with Variance Thresholding	Accuracy with Gaussian Blurring
Sample Image 1	0.699	0.936	-	0.875
Sample Image 2	0.170	0.276	0.315	0.444

Tab. 9 Comparison of OCR accuracies originally vs. before and after variance thresholding

The last method we used to improve OCR was **Adaptive Gaussian Thresholding**. It works by taking a gaussian-weighted sum of the neighbourhood values minus the given constant **C**. This is better than adaptive Otsu since it weighs the values of closer pixels more and guarantees a highly localised threshold. It provided us with the best results - **1.0 for image 1 and 0.649 for image 2**.

S.No	Accuracy without Otsu Thresholding	Accuracy with Adaptive Otsu Thresholding	Accuracy with Variance Thresholding	Accuracy with Gaussian Blurring	Accuracy with Adaptive Gaussian Thresholding
Sample Image 1	0.699	0.936	-	0.875	1.0
Sample Image 2	0.170	0.276	0.315	0.444	0.649

Tab. 10 Comparison of OCR accuracies originally vs. before and after Adaptive Gaussian Thresholding

It was thus concluded that Adaptive Gaussian Thresholding worked the best as it gave the highest OCR accuracy of 100% for Sample Image 1 and 64.9% for Sample Image 2.

Section 3 of the report focused on improving the OCR accuracies of images suffering from various types of image degradation. From the experiments, it was concluded that implementing the right type of deterioration removal techniques always resulted in an increase in OCR accuracy. This tells us how important the pre-processing step is, since most of the images in real word suffer from different types of image degradation which adversely affect the OCR accuracy. **No matter how good the OCR algorithm is, pre-processing steps need to be implemented efficiently for a good OCR accuracy of the overall architecture.**

Appendix A: Python Code

We recommend opening the jupyter notebook submitted to better see the code. The python code extracted from the notebook is as follows:

```
# -*- coding: utf-8 -*-
"""CE4003 - OCR

Automatically generated by Colaboratory.

Original file is located at
   https://colab.research.google.com/drive/1xdEk2aA537ICqsQLLZ9n1epWrr3XZtxl

# Optical Character Recognition

We will be doing the following:

0. Dependency and data setup
1. OCR with Otsu
 i. Creating Otsu algorithm for image binarization
 ii. Feeding Otsu output to Tesseract
2. Improving Otsu:
 i. Using adaptive thresholding
 ii. Background removal using variance
 iii. Background removal using blurring
 iv. Adaptive Gaussian thresholding
3. Image pre-processing for robust OCR
 i.   Perspective transform
 ii.  Skew correction
 iii. Binarization
 iv.  Noise removal
 v.   Thinning

## 0. Dependency and data setup

We will do the following:

i. Install relevant dependencies
ii. Import the images we will be using for testing

### i. Importing dependencies
"""
```

```
! sudo apt install tesseract-ocr
! pip install pytesseract

from google.colab.patches import cv2_imshow
import matplotlib.pyplot as plt
import numpy as np
import pytesseract
import spacy
import math
import cv2

from PIL import ImageFilter, ImageEnhance
import PIL

import nltk
nltk.download('all')

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

"""### ii. Loading the data"""

# Creating function to load image
def load_img(path):
 img = PIL.Image.open(path).convert("L")
 img = np.asarray(img)
 return img

img1 = load_img("/content/sample01.png")
plt.imshow(img1, cmap="gray")

img2 = load_img("/content/sample02.png")
plt.imshow(img2, cmap="gray")

"""## 1. OCR with Otsu

We will now do the following:
i. Define our own function for Otsu thresholding
ii. Utilize PyTesseract for character recognition

### i. Creating custom Otsu function
```

```
"""

# Creating function to check accuracy of OCR using cosine similarity
def accuracy(original, ocr):
 try:
   # Tokenization
   original_tokens = word_tokenize(original)
   ocr_tokens = word_tokenize(ocr)

   # Get list of english stopwords
   stop_words = stopwords.words('english')

   # Extract any stopword present in either string
   original_set = {w for w in original_tokens if not w in stop_words}
   ocr_set = {w for w in ocr_tokens if not w in stop_words}

   original_list =[]
   ocr_list =[]
   # Create a set with keywords from both strings
   union = original_set.union(ocr_set)
   for w in union:
       if w in original_set: original_list.append(1) # Initialize a new vector
       else: original_list.append(0)

       if w in ocr_set: ocr_list.append(1) # Initialize a new vector
       else: ocr_list.append(0)
   c = 0

   # Get cosine similarity
   for i in range(len(union)):
           c += original_list[i] * ocr_list[i]
   cosine = c / float((sum(original_list)*sum(ocr_list))**0.5)

   return cosine
 except:
   return 0.0

# Creating Otsu functions

# To calculate and plot histogram of image
def get_histogram(img):
 hist, bins = np.histogram(img, 256, [0, 256])
```

```
 return hist, bins

def show_histogram(hist, bins):
 width = 0.7 * (bins[1] - bins[0])
 center = (bins[:-1] + bins[1:]) / 2
 plt.bar(center, hist, align='center', width=width)
 plt.show()

# To calculate total number of pixels in histogram
def get_pixel_count(histogram):
 total = 0
 for i in range(len(histogram)):
   total += histogram[i]
 return total

# To get sum of pixel probabilities in a given range
def get_probs(histogram, low, high, pixel_count):
 total = 0
 for r in range(low, high):
   total += histogram[r]/pixel_count
 return total

# To calculate mean of the pixels in a given range
def get_mean(histogram, low, high, pixel_count, probs):
 total = 0
 for r in range(low, high):
   total += (r * histogram[r] / pixel_count)
 total /= probs
 return total

# To calculate variance
def get_variance(histogram, low, high, pixel_count, probs, mean):
 total = 0
 for r in range(low, high):
   total += ((r - mean)**2) * (histogram[r] / pixel_count)
 total /= probs
 return total

# To get weighted class variance
def get_weighted_variance(histogram, low, high, pixel_count):
 q = get_probs(histogram, low, high, pixel_count)
 mean = get_mean(histogram, low, high, pixel_count, q)
```

```
 variance = get_variance(histogram, low, high, pixel_count, q, mean)

 return (variance**2) * q

# To get threshold value which minimzes intra class variance
def get_optimal_threshold(histogram):
 pixel_count = get_pixel_count(histogram)
 optimal_threshold = 0
 lowest_variance = -1
 for threshold in range(1, 256):
   try:
     weighted_variance_L = get_weighted_variance(histogram, 0, threshold, pixel_count)
     weighted_variance_R = get_weighted_variance(histogram, threshold+1, 256,
pixel_count)
     weighted_variance_total = weighted_variance_L + weighted_variance_R
   except:
     continue

   # First iteration or lower variance value
   if lowest_variance == -1 or weighted_variance_total < lowest_variance:
     lowest_variance = weighted_variance_total
     optimal_threshold = threshold

 return optimal_threshold

# To threshold the given image
def threshold_image(img, threshold):
 img = (img > threshold) * 255
 return np.asarray(img, dtype=np.int32)

# Conduct Otsu thresholding on an image
def otsu(img, show_hist=False):
 hist, bins = get_histogram(img)
 threshold = get_optimal_threshold(hist.tolist())
 if show_hist:
   show_histogram(hist, bins)
   print(f"Image threshold: {threshold}")
 output = np.zeros_like(img)
 output[:,:] = threshold_image(img, threshold)
 return output

# Function to display an image
```

```
def display(img):
 plt.imshow(img, cmap="gray")

# Thresholding the first image
otsu_img1 = otsu(img1, True)
display(otsu_img1)

# Thresholding the second image
otsu_img2 = otsu(img2, True)
display(otsu_img2)

"""We can see that the thresholding isn't able to binarize the text properly. This is
because the images contain shadowy regions which skew the histograms and we can also
see that the histograms aren't bimodal which is necessary for good Otsu threshodling.

### ii. Character recognition using PyTesseract
"""

# Noting down original texts for sample images
original_text_1 = "Parking: You may park anywhere on the campus where there are no
signs prohibiting par- \n king. Keep in mind the carpool hours and park accordingly so
you do not get blocked in the afternoon \n\n Under School Age Children:While we love
the younger children, it can be disruptive and \n inappropriate to have them on campus
during school hours. There may be special times \n that they may be invited or can
accompany a parent volunteer, but otherwise we ask \n that you adhere to our policy
for the benefit of the students and staff."
original_text_2 = "Sonnet for Lena\n O dear Lena, your beauty is so vast \n It is hard
sometimes to describe it fast. I thought the entire world I would impress \n If only
your portrait I could compress. \n Alas! First when I tried to use VQ \n I found that
your cheeks belong to only you. \n Your silky hair contains a thousand lines \n Hard
to match with sums of discrete cosines. \n And for your lips, sensual and tactual \n
Thirteen Crays found not the perfect fractal. \n And while these setbacks are all
quite severe \n I might have fixed them with hacks here or there \n But when filters
took sparkle from your eyes \n I said, 'Damn all this. I'll just digitize.' \n Thomas
Colthurst"

# Character recognition without Otsu thresholding
ocr_text_1 = pytesseract.image_to_string(img1)
print(f"First image:")
print(f"{ocr_text_1}")
print(f"Accuracy: {accuracy(original_text_1, ocr_text_1)}")
```

```
# Character recognition without Otsu thresholding
ocr_text_2 = pytesseract.image_to_string(img2)
print(f"Second image:")
print(f"{ocr_text_2}")
print(f"Accuracy: {accuracy(original_text_2, ocr_text_2)}")

# Character recognition with Otsu thresholding
otsu_img1 = otsu(img1)
display(otsu_img1)

ocr_otsu_text_1 = pytesseract.image_to_string(otsu_img1)
print(ocr_otsu_text_1)
print(f"Accuracy: {accuracy(original_text_1, ocr_otsu_text_1)}")

# Character recognition with Otsu thresholding
plt.imshow(otsu_img2, cmap="gray")

ocr_otsu_text_2 = pytesseract.image_to_string(otsu_img2)
print(ocr_otsu_text_2)
print(f"Accuracy: {accuracy(original_text_2, ocr_otsu_text_2)}")

"""__Conclusion:__ The images without thresholding are able to produce average outputs
while those with Otsu give either the same results or no results. This shows that the
basic Otsu algorithm fails in instances of shadows in images or images with varying
lightness.

## 2. Improving Otsu

We will improve the thresholding using the following 2 techniques:
i. Adaptive thresholding
ii. Background removal using variance
iii. Background removal using blurring
iv. Adaptive Gaussian thresholding

### i. OCR with adaptive thresholding

The problem we notice is that the global threshold may be a good global average but
fails in certain areas of the image. Instead of taking a global threshold, we can
break up the image into smaller blocks and conduct Otsu thresholding in these blocks,
which will allow us to learn region-specific thresholds.
"""
```

```
def adaptive_otsu(img, row_block_size, col_block_size):
 assert col_block_size <= img.shape[1] and row_block_size <= img.shape[0]

 output = np.zeros_like(img)

 # Split the image into blocks and run Otsu
 for row in range(0, img.shape[0], row_block_size):
   for col in range(0, img.shape[1], col_block_size):
     row_start, row_end = row, min(img.shape[0], row+row_block_size)
     col_start, col_end = col, min(img.shape[1], col+col_block_size)

     block = img[row_start:row_end, col_start:col_end]
     otsu_block = otsu(block)
     output[row_start:row_end, col_start:col_end] = otsu_block

 return output

# Character recognition with Otsu thresholding
adaptive_otsu_img1 = adaptive_otsu(img1, 100, 200)
display(adaptive_otsu_img1)

ocr_adaptive_otsu_text_1 = pytesseract.image_to_string(adaptive_otsu_img1)
print(ocr_adaptive_otsu_text_1)
print(f"Accuracy: {accuracy(original_text_1, ocr_adaptive_otsu_text_1)}")

# Character recognition with Otsu thresholding
adaptive_otsu_img2 = adaptive_otsu(img2, 50, 50)
display(adaptive_otsu_img2)

ocr_adaptive_otsu_text_2 = pytesseract.image_to_string(adaptive_otsu_img2)
print(ocr_adaptive_otsu_text_2)
print(f"Accuracy: {accuracy(original_text_2, ocr_adaptive_otsu_text_2)}")

"""__Conclusion:__ We see that adaptive thresholding is far superior to the initial
Otsu algorithm for both images. We see that tesseract outputs almost the same text as
there is in image 1. For image 2, we see that most of the text is not accurate since
the thresholded image is very blurry and still noisy. We can also see that the block
size depends on the image characteristics. This is why, we need to pre-process images
before conducting OCR.

### ii. Background removal using variance
```

```
We see in the output for `sample02.png` that many background areas are thresholded to
be black even though they are part of the background, just because of shadows. We can
see that the variance in the small blocks which contain just the background are very
low while the blocks with actual text have higher variance. We use this information to
threshold the background.
"""

def adaptive_otsu_with_bg(img, row_block_size, col_block_size, detect_bg=False,
bg_thresh=50):
 assert col_block_size <= img.shape[1] and row_block_size <= img.shape[0]

 output = np.zeros_like(img)

 # Split the image into blocks and run Otsu
 for row in range(0, img.shape[0], row_block_size):
   for col in range(0, img.shape[1], col_block_size):
     row_start, row_end = row, min(img.shape[0], row+row_block_size)
     col_start, col_end = col, min(img.shape[1], col+col_block_size)

     block = img[row_start:row_end, col_start:col_end]
     otsu_block = otsu(block)

     if detect_bg:
       var = np.var(block)
       cv2_imshow(block)
       print(f"Variance: {var}")
       if var < bg_thresh:
         otsu_block.fill(255) # If variance is low, treat as background

     output[row_start:row_end, col_start:col_end] = otsu_block

 return output

# Character recognition with Otsu thresholding
adaptive_otsu_img2_with_bg = adaptive_otsu_with_bg(img2, 50, 50, True)
display(adaptive_otsu_img2_with_bg)

ocr_adaptive_otsu_text_2_with_bg =
pytesseract.image_to_string(adaptive_otsu_img2_with_bg)
print(ocr_adaptive_otsu_text_2_with_bg)
print(f"Accuracy: {accuracy(original_text_2, ocr_adaptive_otsu_text_2_with_bg)}")
```

```
"""__Conclusion:__ We can see that there is an increase in the accuracy of the OCR and
the thresholded image is also much clearer.

### iii. Background removal using blurring

We can also remove the background of the image which contains the shadows in order to
get just the text on a white background
"""

def remove_bg(img, kernel_size=5):
 blurred_img = cv2.GaussianBlur(img,(kernel_size,kernel_size),0)
 no_bg_img = blurred_img - img
 return no_bg_img

# Character recognition with background removal and Adaptive Otsu
no_bg_img1 = remove_bg(img1, 1001)
remove_bg_img1 = adaptive_otsu(no_bg_img1, 100, 200)
display(remove_bg_img1)

ocr_remove_bg_text_1 = pytesseract.image_to_string(remove_bg_img1)
print(ocr_remove_bg_text_1)
print(f"Accuracy: {accuracy(original_text_1, ocr_remove_bg_text_1)}")

# Showing extracted background
display(cv2.GaussianBlur(img1,(1001,1001),0))

# Character recognition with background removal and Adaptive Otsu
no_bg_img2 = remove_bg(img2, 801)
remove_bg_img2 = adaptive_otsu(no_bg_img2, 100, 200)
display(remove_bg_img2)

ocr_remove_bg_text_2 = pytesseract.image_to_string(remove_bg_img2)
print(ocr_remove_bg_text_2)
print(f"Accuracy: {accuracy(original_text_2, ocr_remove_bg_text_2)}")

# Showing extracted background
display(cv2.GaussianBlur(img2,(801,801),0))

"""__Conclusion:__ We can see that blurring is a better method for background removal
than variance detection.

### iv. Adaptive Gaussian Thresholding
```

```
The adaptive gaussian thresholding algorithm takes a gaussian-weighted sum of the
neighbourhood values minus the given constant __C__. This is better than even adaptive
Otsu since it weighs the values of closer pixels more and guarantees a highly
localised threshold.
"""

def adaptive_gaussian(img, block_size, C):
 thresholded = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY, block_size, C)
 return thresholded

# Character recognition with Adaptive Gaussian thresholding
adaptive_gaussian_img1 = adaptive_gaussian(img1, 13, 9)
display(adaptive_gaussian_img1)

ocr_adaptive_gaussian_text_1 = pytesseract.image_to_string(adaptive_gaussian_img1)
print(ocr_adaptive_gaussian_text_1)
print(f"Accuracy: {accuracy(original_text_1, ocr_adaptive_gaussian_text_1)}")

# Character recognition with Adaptive Gaussian thresholding
adaptive_gaussian_img2 = adaptive_gaussian(img2, 37, 4)
display(adaptive_gaussian_img2)

ocr_adaptive_gaussian_text_2 = pytesseract.image_to_string(adaptive_gaussian_img2)
print(ocr_adaptive_gaussian_text_2)
print(f"Accuracy: {accuracy(original_text_2, ocr_adaptive_gaussian_text_2)}")

"""__Conclusion:__ We can see a drastic improvement in the thresholding as well as the
accuracy of the detected text.

## 3. More robust character recognition

In the real world, the images may suffer from several different types of image
degradation. Even with a really good OCR algorithm, we may get a bad OCR accuracy just
because of the image degradation. To take care of this, image pre-processing must be
done. <br> <br>

These pre-processing steps include

1. **Perspective Transform**: A common problem is that most images are taken at an
angle from the actual plane of the paper. This means that the text is not horizontal
```

```
in the image which can sometimes confuse the OCR algorithm. We will apply automatic perspective transformation to ensure that most images are transformed from a 3D plane to a 2D plane.

2. **Skew Correction**: Image obtained from the previous stage may not be correctly oriented, it may be aligned at any angle. So we need to perform skew correction to make sure that the image forwarded to subsequent stages is correctly oriented.

3. **Binarization:** It means converting a Coloured image to Binary image (containing only black & white colours). Usually, in practice, this conversion of Coloured image to Binary image is done by an intermediate GrayScale image. [This step has already been done using the Otsu thresholding algorithm]

4. **Speckle Noise Removal:** Noise (small dots or foreground components) may be introduced easily into an image while scanning it during Image Acquisition because of various reasons like low clarity camera, a shadow on image etc. This noise should be removed so that the image will be clean and uniform.

5. **Thinning:** Different images have text with different width of strokes. This variability is very high in the case of handwritten words. Skeletonization is a technique, using which we can make all strokes to have a uniform width (Maybe 1 pixel wide or few pixels wide)

### 3.1 Perspective transform
"""

# Loading a 3D perspective image
paper = load_img("/content/paper.jpeg")
display(paper)

# Noting down original text for image
original_text_paper = "This paper contains 2 questions and comprises 2 pages. \n Answer ALL questions. \n This is a closed-book examination. \n This paper carries 30 marks"

# Conducting thresholding and OCR on image
adaptive_otsu_paper = adaptive_otsu(paper, 100, 200)
display(adaptive_otsu_paper)

ocr_adaptive_otsu_paper = pytesseract.image_to_string(adaptive_otsu_paper)
print(ocr_adaptive_otsu_paper)
print(f"Accuracy: {accuracy(original_text_paper, ocr_adaptive_otsu_paper)}")
```

```
def perspective_transform(img, corners):
 (tl, tr, br, bl) = corners

 # Compute width of new image
 width_top = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2)) # Distance
between top right and left
 width_bottom = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2)) # Distance
between bottom right and left
 width = max(int(width_top), int(width_bottom)) # Maximum of above 2

 # Compute height of new image
 height_left = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2)) # Distance
between left top and bottom
 height_right = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2)) # Distance
between right top and bottom
 height = max(int(height_right), int(height_left))

 # Creating points to map corner points to in new image
 corners_new = np.array([
   [0, 0],
   [width - 1, 0],
   [width - 1, height - 1],
   [0, height - 1]], dtype = "float32")
 # Compute perspective transform matrix and transform image
 transform_matrix = cv2.getPerspectiveTransform(corners, corners_new)
 transformed_image = cv2.warpPerspective(img, transform_matrix, (width, height))

 return transformed_image

paper_corners = np.asarray([(100, 10), (1600, 180), (1600, 550), (100, 600)], dtype =
"float32")
transformed_paper = perspective_transform(paper, paper_corners)
display(transformed_paper)

# Conducting thresholding and OCR on image
adaptive_otsu_paper_transformed = adaptive_otsu(transformed_paper, 100, 200)
display(adaptive_otsu_paper_transformed)

ocr_adaptive_otsu_paper_transformed =
pytesseract.image_to_string(adaptive_otsu_paper_transformed)
print(ocr_adaptive_otsu_paper_transformed)
```

```
print(f"Accuracy: {accuracy(original_text_paper,
ocr_adaptive_otsu_paper_transformed)}")

"""__Conclusion:__ As we can see, the image without the transformation was not
recognised properly since the OCR algorithm is created for 2D planar text. After we
conducted perspective transform on the image, the recognised text is a lot more
accurate.

### 3.2 Image de-skewing
"""

#Loading a 2D-skewed image
skewed_paper = load_img("/content/skewedpaper.png")
display(skewed_paper)

# Noting down original text for image
original_text_skewed = "Our last argument is how we want to approximate the contour.
We use cv2.CHAIN_APPROX_SIMPLE to compress horizontal, vertical, and diagonal segments
into their endpoints only. This saves both computation and memory. If we wanted all
the points along the contour, without compression, we can pass in
cv2.CHAIN_APPROX_NONE; however, be very sparing when using this function. Retrieving
all points along a contour is often unnecessary and is wasteful of resources."

# Conducting thresholding and OCR on image
adaptive_otsu_paper_skewed = adaptive_otsu(skewed_paper, 300, 300)
display(adaptive_otsu_paper_skewed)

ocr_adaptive_otsu_paper_skewed =
pytesseract.image_to_string(adaptive_otsu_paper_skewed)
print(ocr_adaptive_otsu_paper_skewed)
print(f"Accuracy: {accuracy(original_text_skewed, ocr_adaptive_otsu_paper_skewed)}")

#Defining the deskew functions

# Calculate skew angle of an image
def getSkewAngle(cvImage):
    # Prep image, copy, convert to gray scale, blur, and threshold
   newImage = cvImage.copy()
   blur = cv2.GaussianBlur(newImage, (9, 9), 0)
   thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)[1]

   # Apply dilate to merge text into meaningful lines/paragraphs.
```

```
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (30, 5))
    dilate = cv2.dilate(thresh, kernel, iterations=5)

    # Find all contours
    contours, hierarchy = cv2.findContours(dilate, cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)
    contours = sorted(contours, key = cv2.contourArea, reverse = True)

    # Find largest contour and surround in min area box
    largestContour = contours[0]
    minAreaRect = cv2.minAreaRect(largestContour)

    # Determine the angle. Convert it to the value that was originally used to obtain
skewed image
    angle = minAreaRect[-1]
    if angle < -45:
        angle = 90 + angle
    return -1.0 * angle


# Rotate the image around its center
def rotateImage(cvImage, angle: float):
    newImage = cvImage.copy()
    (h, w) = newImage.shape[:2]
    center = (w // 2, h // 2)
    M = cv2.getRotationMatrix2D(center, angle, 1.0)
    newImage = cv2.warpAffine(newImage, M, (w, h), flags=cv2.INTER_CUBIC,
borderMode=cv2.BORDER_REPLICATE)
    return newImage

# Deskew image
def deskew(cvImage):
    angle = getSkewAngle(cvImage)
    return rotateImage(cvImage, -1.0 * angle)

deskewed_paper = deskew(skewed_paper)
display(deskewed_paper)

# Conducting thresholding and OCR on image
adaptive_otsu_paper_deskewed = adaptive_otsu(deskewed_paper, 300, 300)
display(adaptive_otsu_paper_deskewed)
```

```
ocr_adaptive_otsu_paper_deskewed =
pytesseract.image_to_string(adaptive_otsu_paper_deskewed)
print(ocr_adaptive_otsu_paper_deskewed)
print(f"Accuracy: {accuracy(original_text_skewed, ocr_adaptive_otsu_paper_deskewed)}")

"""### 3.4 Removing Speckle Noise


"""

#Loading a speckled image
speckle = load_img("/content/noise2.png")
display(speckle)

# Noting down original text for image
original_text_noise = "Hi, my name is Madhav, this is a test for my optical character
recognition algorithm; please work"

# Conducting thresholding and OCR on image
adaptive_otsu_speckle = adaptive_otsu(speckle, 250, 250)
display(adaptive_otsu_speckle)

ocr_adaptive_otsu_speckle = pytesseract.image_to_string(adaptive_otsu_speckle)
print(ocr_adaptive_otsu_speckle)
print(f"Accuracy: {accuracy(original_text_noise, ocr_adaptive_otsu_speckle)}")

#Removing noise using cv2 gaussian filter
gaussian_speckle = cv2.GaussianBlur(speckle, (9, 9), 0)

# Plotting of image after Gaussian Filter
display(gaussian_speckle)
plt.show()

# Conducting thresholding and OCR on image
adaptive_otsu_gaussian = adaptive_otsu(gaussian_speckle, 250, 250)
display(adaptive_otsu_gaussian)

ocr_adaptive_otsu_gaussian = pytesseract.image_to_string(adaptive_otsu_gaussian)
print(ocr_adaptive_otsu_gaussian)
print(f"Accuracy: {accuracy(original_text_noise, ocr_adaptive_otsu_gaussian)}")
```

```
"""As it can be clearly seen, the output after removing noise using the Gaussian
filter is much clearer than without removing it.

### 3.5 Thinning

Thinning in OCR is required mostlly with handwritten documents, when the text is too
thick and thus some characters overlap with the others. Thinning reduces the width of
each character, thus reducing the overlap and making it easier for the OCR algorithm
to detect indivudual characters
"""

#Loading a speckled image
comp1 = load_img("/content/cz40032.png")
# comp1 = cv2.rotate(comp1, cv2.ROTATE_90_COUNTERCLOCKWISE)
plt.imshow(comp1, cmap="gray")

# Noting down original text for image
original_text_thin = "CZ 4003"

# Conducting thresholding and OCR on image
adaptive_otsu_thin = adaptive_otsu(comp1, 50, 50)
display(adaptive_otsu_thin)

ocr_adaptive_otsu_thin = pytesseract.image_to_string(adaptive_otsu_thin)
print(ocr_adaptive_otsu_thin)
print(f"Accuracy: {accuracy(original_text_thin, ocr_adaptive_otsu_thin)}")

kernel = np.ones((7,7),np.uint8)
thinned_comp1 = cv2.erode(comp1, kernel,iterations = 1)

display(thinned_comp1)

# Conducting thresholding and OCR on image
adaptive_otsu_thinned = adaptive_otsu(thinned_comp1, 50, 50)
display(adaptive_otsu_thinned)

ocr_adaptive_otsu_thinned = pytesseract.image_to_string(adaptive_otsu_thinned)
print(ocr_adaptive_otsu_thinned)
print(f"Accuracy: {accuracy(original_text_thin, ocr_adaptive_otsu_thinned)}")
```

Appendix B: Jupyter Notebook

The jupyter notebook can be run by uploading to Google Colaboratory alongside the submitted images. A screenshot of the jupyter notebook has been appended to this report starting from the next page:

Optical Character Recognition

We will be doing the following:

0. Dependency and data setup
1. OCR with Otsu
   i. Creating Otsu algorithm for image binarization
   ii. Feeding Otsu output to Tesseract
2. Improving Otsu:
   i. Using adaptive thresholding
   ii. Background removal using variance
   iii. Background removal using blurring
   iv. Adaptive Gaussian thresholding
3. Image pre-processing for robust OCR
   i. Perspective transform
   ii. Skew correction
   iii. Binarization
   iv. Noise removal
   v. Thinning

0. Dependency and data setup

We will do the following:

i. Install relevant dependencies
ii. Import the images we will be using for testing

i. Importing dependencies

[1]
```
! sudo apt install tesseract-ocr
! pip install pytesseract
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  tesseract-ocr-eng tesseract-ocr-osd
The following NEW packages will be installed:
  tesseract-ocr tesseract-ocr-eng tesseract-ocr-osd
0 upgraded, 3 newly installed, 0 to remove and 14 not upgraded.
Need to get 4,795 kB of archives.
After this operation, 15.8 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu bionic/universe amd64 tesseract-ocr-eng all 4.00~git24-0e00fe6-1.2 [1,588 kB]
Get:2 http://archive.ubuntu.com/ubuntu bionic/universe amd64 tesseract-ocr-osd all 4.00~git24-0e00fe6-1.2 [2,989 kB]
Get:3 http://archive.ubuntu.com/ubuntu bionic/universe amd64 tesseract-ocr amd64 4.00~git2288-10f4998a-2 [218 kB]
Fetched 4,795 kB in 3s (1,668 kB/s)
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm line 76, <> line 3.)
debconf: falling back to frontend: Readline
debconf: unable to initialize frontend: Readline
debconf: (This frontend requires a controlling tty.)
debconf: falling back to frontend: Teletype
dpkg-preconfigure: unable to re-open stdin:
Selecting previously unselected package tesseract-ocr-eng.
(Reading database ... 144793 files and directories currently installed.)
Preparing to unpack .../tesseract-ocr-eng_4.00~git24-0e00fe6-1.2_all.deb ...
Unpacking tesseract-ocr-eng (4.00~git24-0e00fe6-1.2) ...
Selecting previously unselected package tesseract-ocr-osd.
Preparing to unpack .../tesseract-ocr-osd_4.00~git24-0e00fe6-1.2_all.deb ...
Unpacking tesseract-ocr-osd (4.00~git24-0e00fe6-1.2) ...
Selecting previously unselected package tesseract-ocr.
Preparing to unpack .../tesseract-ocr_4.00~git2288-10f4998a-2_amd64.deb ...
Unpacking tesseract-ocr (4.00~git2288-10f4998a-2) ...
Setting up tesseract-ocr-osd (4.00~git24-0e00fe6-1.2) ...
Setting up tesseract-ocr-eng (4.00~git24-0e00fe6-1.2) ...
Setting up tesseract-ocr (4.00~git2288-10f4998a-2) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Collecting pytesseract
  Downloading https://files.pythonhosted.org/packages/17/4b/4dbd55388225bb6cd243d21f70e77cb3ce061e241257485936324b8e920f/pytesseract-0.3.6.tar.gz
Requirement already satisfied: Pillow in /usr/local/lib/python3.6/dist-packages (from pytesseract) (7.0.0)
Building wheels for collected packages: pytesseract
  Building wheel for pytesseract (setup.py) ... done
  Created wheel for pytesseract: filename=pytesseract-0.3.6-py2.py3-none-any.whl size=13629 sha256=69a800aad5d884b9cad6a9f62e66e8e9540e69bf92a35904b5b58ab1f5aa2a69
  Stored in directory: /root/.cache/pip/wheels/ee/71/72/b98430261d849ae631e283dfc7ccb456a3fb2ed2205714b63f
Successfully built pytesseract
Installing collected packages: pytesseract
Successfully installed pytesseract-0.3.6
```

[2]
```
from google.colab.patches import cv2_imshow
import matplotlib.pyplot as plt
import numpy as np
import pytesseract
import spacy
import math
import cv2

from PIL import ImageFilter, ImageEnhance
import PIL

import nltk
nltk.download('all')

from nltk.corpus import stopwords
```

```
from nltk.tokenize import word_tokenize
```

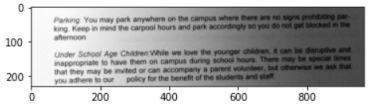
```
[nltk_data]    | Downloading package universal_tagset to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping taggers/universal_tagset.zip.
[nltk_data]    | Downloading package maxent_ne_chunker to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping chunkers/maxent_ne_chunker.zip.
[nltk_data]    | Downloading package punkt to /root/nltk_data...
[nltk_data]    |   Unzipping tokenizers/punkt.zip.
[nltk_data]    | Downloading package book_grammars to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping grammars/book_grammars.zip.
[nltk_data]    | Downloading package sample_grammars to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping grammars/sample_grammars.zip.
[nltk_data]    | Downloading package spanish_grammars to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping grammars/spanish_grammars.zip.
[nltk_data]    | Downloading package basque_grammars to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping grammars/basque_grammars.zip.
[nltk_data]    | Downloading package large_grammars to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping grammars/large_grammars.zip.
[nltk_data]    | Downloading package tagsets to /root/nltk_data...
[nltk_data]    |   Unzipping help/tagsets.zip.
[nltk_data]    | Downloading package snowball_data to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    | Downloading package bllip_wsj_no_aux to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping models/bllip_wsj_no_aux.zip.
[nltk_data]    | Downloading package word2vec_sample to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping models/word2vec_sample.zip.
[nltk_data]    | Downloading package panlex_swadesh to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    | Downloading package mte_teip5 to /root/nltk_data...
[nltk_data]    |   Unzipping corpora/mte_teip5.zip.
[nltk_data]    | Downloading package averaged_perceptron_tagger to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping taggers/averaged_perceptron_tagger.zip.
[nltk_data]    | Downloading package averaged_perceptron_tagger_ru to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping
[nltk_data]    |       taggers/averaged_perceptron_tagger_ru.zip.
[nltk_data]    | Downloading package perluniprops to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping misc/perluniprops.zip.
[nltk_data]    | Downloading package nonbreaking_prefixes to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    |   Unzipping corpora/nonbreaking_prefixes.zip.
[nltk_data]    | Downloading package vader_lexicon to
[nltk_data]    |     /root/nltk_data...
[nltk_data]    | Downloading package porter_test to /root/nltk_data...
[nltk_data]    |   Unzipping stemmers/porter_test.zip.
[nltk_data]    | Downloading package wmt15_eval to /root/nltk_data...
[nltk_data]    |   Unzipping models/wmt15_eval.zip.
[nltk_data]    | Downloading package mwa_ppdb to /root/nltk_data...
[nltk_data]    |   Unzipping misc/mwa_ppdb.zip.
[nltk_data]    |
[nltk_data]  Done downloading collection all
```

ii. Loading the data

```
[3]  # Creating function to load image
     def load_img(path):
       img = PIL.Image.open(path).convert("L")
       img = np.asarray(img)
       return img
```

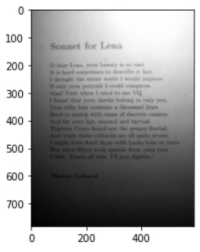
```
[4]  img1 = load_img("/content/sample01.png")
     plt.imshow(img1, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7fa284790be0>
```




```
[5]  img2 = load_img("/content/sample02.png")
     plt.imshow(img2, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7fa284f807b8>
```



1. OCR with Otsu

We will now do the following:

i. Define our own function for Otsu thresholding

ii. Utilize PyTesseract for character recognition

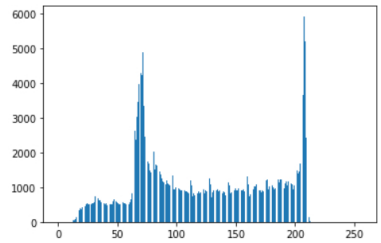
i. Creating custom Otsu function

[6]
```
# Creating function to check accuracy of OCR using cosine similarity
def accuracy(original, ocr):
  try:
    # Tokenization
    original_tokens = word_tokenize(original)
    ocr_tokens = word_tokenize(ocr)

    # Get list of english stopwords
    stop_words = stopwords.words('english')

    # Extract any stopword present in either string
    original_set = {w for w in original_tokens if not w in stop_words}
    ocr_set = {w for w in ocr_tokens if not w in stop_words}

    original_list =[]
    ocr_list =[]
    # Create a set with keywords from both strings
    union = original_set.union(ocr_set)
    for w in union:
        if w in original_set: original_list.append(1) # Initialize a new vector
        else: original_list.append(0)

        if w in ocr_set: ocr_list.append(1) # Initialize a new vector
        else: ocr_list.append(0)
    c = 0

    # Get cosine similarity
    for i in range(len(union)):
            c += original_list[i] * ocr_list[i]
    cosine = c / float((sum(original_list)*sum(ocr_list))**0.5)

    return cosine
  except:
    return 0.0
```

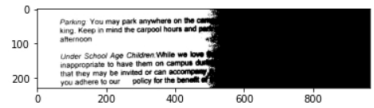
[7]
```
# Creating Otsu functions

# To calculate and plot histogram of image
def get_histogram(img):
  hist, bins = np.histogram(img, 256, [0, 256])
  return hist, bins

def show_histogram(hist, bins):
  width = 0.7 * (bins[1] - bins[0])
  center = (bins[:-1] + bins[1:]) / 2
  plt.bar(center, hist, align='center', width=width)
  plt.show()

# To calculate total number of pixels in histogram
def get_pixel_count(histogram):
  total = 0
  for i in range(len(histogram)):
    total += histogram[i]
  return total

# To get sum of pixel probabilities in a given range
def get_probs(histogram, low, high, pixel_count):
  total = 0
  for r in range(low, high):
    total += histogram[r]/pixel_count
  return total

# To calculate mean of the pixels in a given range
def get_mean(histogram, low, high, pixel_count, probs):
  total = 0
  for r in range(low, high):
    total += (r * histogram[r] / pixel_count)
  total /= probs
  return total

# To calculate variance
def get_variance(histogram, low, high, pixel_count, probs, mean):
  total = 0
  for r in range(low, high):
    total += ((r - mean)**2) * (histogram[r] / pixel_count)
  total /= probs
  return total

# To get weighted class variance
def get_weighted_variance(histogram, low, high, pixel_count):
  q = get_probs(histogram, low, high, pixel_count)
  mean = get_mean(histogram, low, high, pixel_count, q)
  variance = get_variance(histogram, low, high, pixel_count, q, mean)

  return (variance**2) * q

# To get threshold value which minimzes intra class variance
def get_optimal_threshold(histogram):
  pixel_count = get_pixel_count(histogram)
  optimal_threshold = 0
  lowest_variance = -1
  for threshold in range(1, 256):
    try:
      weighted_variance_L = get_weighted_variance(histogram, 0, threshold, pixel_count)
      weighted_variance_R = get_weighted_variance(histogram, threshold+1, 256, pixel_count)
      weighted_variance_total = weighted_variance_L + weighted_variance_R
    except:
      continue

    # First iteration or lower variance value
    if lowest_variance == -1 or weighted_variance_total < lowest_variance:
      lowest_variance = weighted_variance_total
      optimal_threshold = threshold
```

```
    return optimal_threshold

  # To threshold the given image
  def threshold_image(img, threshold):
    img = (img > threshold) * 255
    return np.asarray(img, dtype=np.int32)

  # Conduct Otsu thresholding on an image
  def otsu(img, show_hist=False):
    hist, bins = get_histogram(img)
    threshold = get_optimal_threshold(hist.tolist())
    if show_hist:
      show_histogram(hist, bins)
      print(f"Image threshold: {threshold}")
    output = np.zeros_like(img)
    output[:,:] = threshold_image(img, threshold)
    return output

  # Function to display an image
  def display(img):
    plt.imshow(img, cmap="gray")
```

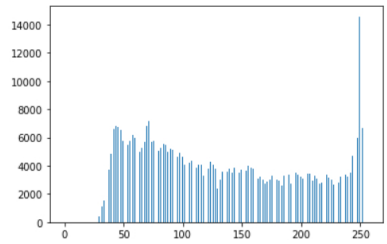
```
[8] # Thresholding the first image
    otsu_img1 = otsu(img1, True)
    display(otsu_img1)
```



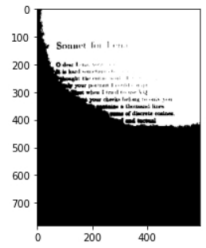
```
Image threshold: 127
```



```
[9] # Thresholding the second image
    otsu_img2 = otsu(img2, True)
    display(otsu_img2)
```



```
Image threshold: 143
```



We can see that the thresholding isn't able to binarize the text properly. This is because the images contain shadowy regions which skew the histograms and we can also see that the histograms aren't bimodal which is necessary for good Otsu threshodling.

ii. Character recognition using PyTesseract

```
[10] # Noting down original texts for sample images
     original_text_1 = "Parking: You may park anywhere on the campus where there are no signs prohibiting par- \n king. Keep in mind the carpool hours and park accordingly so you do not get
     original_text_2 = "Sonnet for Lena\n O dear Lena, your beauty is so vast \n It is hard sometimes to describe it fast. I thought the entire world I would impress \n If only your portrai
```

```
[11] # Character recognition without Otsu thresholding
     ocr_text_1 = pytesseract.image_to_string(img1)
     print(f"First image:")
     print(f"{ocr_text_1}")
     print(f"Accuracy: {accuracy(original_text_1, ocr_text_1)}")
```

```
First image:
Parking: You may park anywhere on the ce
king. Keep in mind the carpool hours and park
afternoon

Under School Age Children:While we love
inappropriate to have them on campus @ )
```

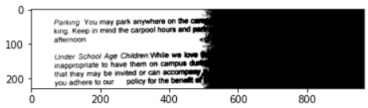
```
that they may be invited or can accompany :
you adhere to our _ policy for the benefit of


Accuracy: 0.6998964726756152
```

```
[12] # Character recognition without Otsu thresholding
     ocr_text_2 = pytesseract.image_to_string(img2)
     print(f"Second image:")
     print(f"{ocr_text_2}")
     print(f"Accuracy: {accuracy(original_text_2, ocr_text_2)}")
```

```
Second image:
Sonnet for Lena


Accuracy: 0.17025130615174972
```

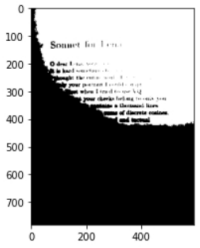
```
[13] # Character recognition with Otsu thresholding
     otsu_img1 = otsu(img1)
     display(otsu_img1)

     ocr_otsu_text_1 = pytesseract.image_to_string(otsu_img1)
     print(ocr_otsu_text_1)
     print(f"Accuracy: {accuracy(original_text_1, ocr_otsu_text_1)}")
```

```
Parking: You may park anywhere on the can
king. Keep in mind the carpool hours and p
afternoon

Under School Age Children:While we love §
inappropriate to have them on campus Gurm
that they may be invited or can accompany,

you adhere to our —_policy for the benefit


Accuracy: 0.6998964726756152
```



```
[14] # Character recognition with Otsu thresholding
     plt.imshow(otsu_img2, cmap="gray")

     ocr_otsu_text_2 = pytesseract.image_to_string(otsu_img2)
     print(ocr_otsu_text_2)
     print(f"Accuracy: {accuracy(original_text_2, ocr_otsu_text_2)}")
```

```
Perna


Accuracy: 0.0
```



Conclusion: The images without thresholding are able to produce average outputs while those with Otsu give either the same results or no results. This shows that the basic Otsu algorithm fails in instances of shadows in images or images with varying lightness.

2. Improving Otsu

We will improve the thresholding using the following 2 techniques:
i. Adaptive thresholding
ii. Background removal using variance
iii. Background removal using blurring
iv. Adaptive Gaussian thresholding

i. OCR with adaptive thresholding

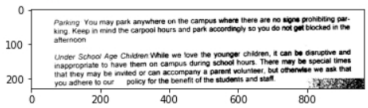
The problem we notice is that the global threshold may be a good global average but fails in certain areas of the image. Instead of taking a global threshold, we can break up the image into smaller blocks and conduct Otsu thresholding in these blocks, which will allow us to learn region-specific thresholds.

```
[15] def adaptive_otsu(img, row_block_size, col_block_size):
```

```
    assert col_block_size <= img.shape[1] and row_block_size <= img.shape[0]

    output = np.zeros_like(img)

    # Split the image into blocks and run Otsu
    for row in range(0, img.shape[0], row_block_size):
      for col in range(0, img.shape[1], col_block_size):
        row_start, row_end = row, min(img.shape[0], row+row_block_size)
        col_start, col_end = col, min(img.shape[1], col+col_block_size)

        block = img[row_start:row_end, col_start:col_end]
        otsu_block = otsu(block)
        output[row_start:row_end, col_start:col_end] = otsu_block

    return output
```

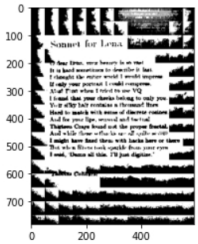
```
[16] # Character recognition with Otsu thresholding
     adaptive_otsu_img1 = adaptive_otsu(img1, 100, 200)
     display(adaptive_otsu_img1)

     ocr_adaptive_otsu_text_1 = pytesseract.image_to_string(adaptive_otsu_img1)
     print(ocr_adaptive_otsu_text_1)
     print(f"Accuracy: {accuracy(original_text_1, ocr_adaptive_otsu_text_1)}")
```

```
Parking: You may park anywhere on the campus where there are no signe prohibiting par-
king. Keep in mind the carpool hours and park accordingly so you do not get blocked in the
afternoon

Under Schoo! Age Children:While we love the younger children, it can be disruptive and
inappropriate to have them on campus during school hours. There may be special times
that they may be invited or can accompany a parent volunteer, but otherwise we ask that
you adhere to our —_ policy for the benefit of the students and staff.


Accuracy: 0.9363821838346235
```



```
[17] # Character recognition with Otsu thresholding
     adaptive_otsu_img2 = adaptive_otsu(img2, 50, 50)
     display(adaptive_otsu_img2)

     ocr_adaptive_otsu_text_2 = pytesseract.image_to_string(adaptive_otsu_img2)
     print(ocr_adaptive_otsu_text_2)
     print(f"Accuracy: {accuracy(original_text_2, ocr_adaptive_otsu_text_2)}")
```

```
h, Aer Dena, come benuty in so rant itediaaitsdll
Ty ig bard sounetiznes to deseribe it inst. ala

Tthaght the entire world} eculd impress
Lf omly vour portrait [ crak] rompreee.
Atagl! Pieat when f tricct tt tse VQ
L fomod that your checks belkng to ouly you.
Pulr alike hal? contains a Mion thee

i Herd to match with muitos of discrete cosines |
Aud for your fips, sensual and factual
Thigtees Crays found wut ibe proper fractal
And stile then sefhacks are all quite 9st

.. A might have fixed thea: with hacks here or there ,

Bait when Aliem took spagkle from your eves

Dandd, ‘Demo all thie. I'l just digitize.” mad


Accuracy: 0.27612404178634276
```



Conclusion: We see that adaptive thresholding is far superior to the initial Otsu algorithm for both images. We see that tesseract outputs almost the same text as there is in image 1. For image 2, we see that most of the text is not accurate since the thresholded image is very blurry and still noisy. We can also see that the block size depends on the image characteristics. This is why, we need to pre-process images before conducting OCR.

ii. Background removal using variance

We see in the output for `sample02.png` that many background areas are thresholded to be black even though they are part of the background, just because of shadows. We can see that the variance in the small blocks which contain just the background are very low while the blocks with actual text have higher variance. We use this information to threshold the background.

```
[18] def adaptive_otsu_with_bg(img, row_block_size, col_block_size, detect_bg=False, bg_thresh=50):
       assert col_block_size <= img.shape[1] and row_block_size <= img.shape[0]
```

```
    output = np.zeros_like(img)

    # Split the image into blocks and run Otsu
    for row in range(0, img.shape[0], row_block_size):
      for col in range(0, img.shape[1], col_block_size):
        row_start, row_end = row, min(img.shape[0], row+row_block_size)
        col_start, col_end = col, min(img.shape[1], col+col_block_size)

        block = img[row_start:row_end, col_start:col_end]
        otsu_block = otsu(block)

        if detect_bg:
          var = np.var(block)
          cv2_imshow(block)
          print(f"Variance: {var}")
          if var < bg_thresh:
            otsu_block.fill(255) # If variance is low, treat as background

        output[row_start:row_end, col_start:col_end] = otsu_block

    return output
```

```
[19] # Character recognition with Otsu thresholding
     adaptive_otsu_img2_with_bg = adaptive_otsu_with_bg(img2, 50, 50, True)
     display(adaptive_otsu_img2_with_bg)

     ocr_adaptive_otsu_text_2_with_bg = pytesseract.image_to_string(adaptive_otsu_img2_with_bg)
     print(ocr_adaptive_otsu_text_2_with_bg)
     print(f"Accuracy: {accuracy(original_text_2, ocr_adaptive_otsu_text_2_with_bg)}")
```

```
Variance: 4.04174375
```

```
Variance: 3.5701734375000003
```

```
Variance: 3.9712437499999993
```

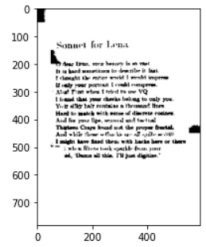
```
Variance: 3.7735999999999996
```

```
Variance: 4.3851234375
```

```
Variance: 4.534960937500001
```

```
Variance: 3.6893109375000006
```

```
Variance: 3.825256307527943
ynnel for Lena


h Aer Dena, come benuty in so rant
Hy ie bard soineti:nes to deecribe it Laat.
Tthaght the entire world} eculd impress
Lf omly vour portrait [ crak] rompreee.
Atagl! Pieat when f tricct tt tse VQ
L fomod that your checks belkng to ouly you.
Pulr alike hal? contains a Mion thee
Hard to match with mums of discrete cosines.
Aud for your fips, sensual and factual
Thigtees Crays found wut ibe proper fractal «
And stile fhene sefhackc are all qaike ice
i might have fixed thea: with hacks here or there
6 when filiem took sparkle frou: your

wd, ‘Demo all this. I'l) just digitize.”

Accuracy: 0.31549666086411104
```



Conclusion: We can see that there is an increase in the accuracy of the OCR and the thresholded image is also much clearer.

iii. Background removal using blurring

We can also remove the background of the image which contains the shadows in order to get just the text on a white background

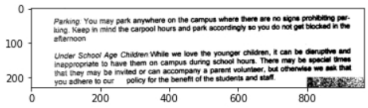
```
[20] def remove_bg(img, kernel_size=5):
       blurred_img = cv2.GaussianBlur(img,(kernel_size,kernel_size),0)
       no_bg_img = blurred_img - img
       return no_bg_img
```

```
[21] # Character recognition with background removal and Adaptive Otsu
     no_bg_img1 = remove_bg(img1, 1001)
     remove_bg_img1 = adaptive_otsu(no_bg_img1, 100, 200)
     display(remove_bg_img1)

     ocr_remove_bg_text_1 = pytesseract.image_to_string(remove_bg_img1)
     print(ocr_remove_bg_text_1)
     print(f"Accuracy: {accuracy(original_text_1, ocr_remove_bg_text_1)}")
```

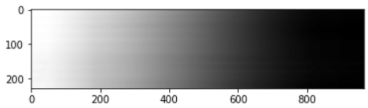
```
print(f"Accuracy: {accuracy(original_text_1, ocr_remove_bg_text_1)}")
```

```
Parking: You may park anywhere on the campus where there are no signs prohibiting par-
king. Keep in mind the carpool hours and park accordingly $0 you do not get blocked in the
afternoon

Under Schoo! Age Children:While we love the younger children, it can be disruptive and
inappropriate to have them on campus during schoo! hours. There may be special times
that they may be invited or can accompany 4 parent volunteer, but otherwise we oak that
you adhere to our —_policy for the benefit of the students and staff.
```

```
Accuracy: 0.8757605390397141
```



[22]
```
# Showing extracted background
display(cv2.GaussianBlur(img1,(1001,1001),0))
```

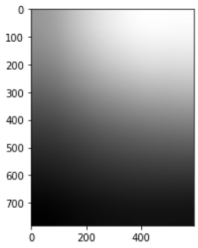


[23]
```
# Character recognition with background removal and Adaptive Otsu
no_bg_img2 = remove_bg(img2, 801)
remove_bg_img2 = adaptive_otsu(no_bg_img2, 100, 200)
display(remove_bg_img2)

ocr_remove_bg_text_2 = pytesseract.image_to_string(remove_bg_img2)
print(ocr_remove_bg_text_2)
print(f"Accuracy: {accuracy(original_text_2, ocr_remove_bg_text_2)}")
```

```
O deer Lena, vour fesuty in ao vent

Te ia bard sanelinnes ta clescribe it baat,
Pthought the entire work! } wand impress
Hf only your portrait [euuhd comprenn.

Alas! Fire when I cciedt to nee VQ

[ fowod that your cheeks belong to only you.
Your silky bar coptainn @ thoussod lines
Hard to match with sume of discrete cosines.
And for your lips, setewal aid tactuad
Thirtern Crays found not the proper fractal,
And while thee setbacks arr all quile severe
I might have fixed them with hacks here or thers
Bet when filtem Look sparkle from your eyes
Deaid, ‘Doma all thie. I'll just digitive.”
```

```
Accuracy: 0.44416506919798654
```

[24]
```
# Showing extracted background
display(cv2.GaussianBlur(img2,(801,801),0))
```



Conclusion: We can see that blurring is a better method for background removal than variance detection.

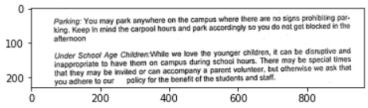
iv. Adaptive Gaussian Thresholding

The adaptive gaussian thresholding algorithm takes a gaussian-weighted sum of the neighbourhood values minus the given constant **C**. This is better than even adaptive Otsu since it weighs the values of closer pixels more and guarantees a highly localised threshold.

[25]
```
def adaptive_gaussian(img, block_size, C):
    thresholded = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, block_size, C)
    return thresholded
```

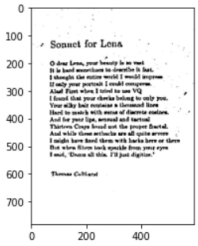
```
[26] # Character recognition with Adaptive Gaussian thresholding
     adaptive_gaussian_img1 = adaptive_gaussian(img1, 13, 9)
     display(adaptive_gaussian_img1)

     ocr_adaptive_gaussian_text_1 = pytesseract.image_to_string(adaptive_gaussian_img1)
     print(ocr_adaptive_gaussian_text_1)
     print(f"Accuracy: {accuracy(original_text_1, ocr_adaptive_gaussian_text_1)}")
```

```
Parking: You may park anywhere on the campus where there are no signs prohibiting par-
king. Keep in mind the carpool hours and park accordingly so you do not get blocked in the
afternoon

Under School Age Children:While we love the younger children, it can be disruptive and
inappropriate to have them on campus during school hours. There may be special times
that they may be invited or can accompany a parent volunteer, but otherwise we ask that
you adhere to our policy for the benefit of the students and staff.

Accuracy: 1.0
```




```
[27] # Character recognition with Adaptive Gaussian thresholding
     adaptive_gaussian_img2 = adaptive_gaussian(img2, 37, 4)
     display(adaptive_gaussian_img2)

     ocr_adaptive_gaussian_text_2 = pytesseract.image_to_string(adaptive_gaussian_img2)
     print(ocr_adaptive_gaussian_text_2)
     print(f"Accuracy: {accuracy(original_text_2, ocr_adaptive_gaussian_text_2)}")
```

```
« Sonnet for Lena , °

O dear Lena, your benuty Ja oa vant

It is hard eometi:nes to deacribe It fast. |

L thought Lhe entire world [ would impress.

Lf onty your portrait [ could compress.

Alas! First when I tried to use VQ

L found that your checks belong to only you.
Your silky bait containa a thousand lines
Hard to match with sums of discrete cosines.
And for your lips, senrual and tactual

Thirteen Craya found not the proper fractal,
And while these setbacks are all quite severe
IT ohight bave fixed them with hacks bere or there
But when filters Look sprrkle from your eyes
Jeni, ‘Damo all thin. I'll Just digitize.”

Thomas Caltharst

Accuracy: 0.6490770196350103
```



Conclusion: We can see a drastic improvement in the thresholding as well as the accuracy of the detected text.

3. More robust character recognition

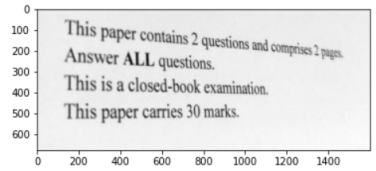
In the real world, the images may suffer from several different types of image degradation. Even with a really good OCR algorithm, we may get a bad OCR accuracy just because of the image degradation. To take care of this, image pre-processing must be done.

These pre-processing steps include

1. **Perspective Transform**: A common problem is that most images are taken at an angle from the actual plane of the paper. This means that the text is not horizontal in the image which can sometimes confuse the OCR algorithm. We will apply automatic perspective transformation to ensure that most images are transformed from a 3D plane to a 2D plane.
2. **Skew Correction**: Image obtained from the previous stage may not be correctly oriented, it may be aligned at any angle. So we need to perform skew correction to make sure that the image forwarded to subsequent stages is correctly oriented.
3. **Binarization:** It means converting a Coloured image to Binary image (containing only black & white colours). Usually, in practice, this conversion of Coloured image to Binary image is done by an intermediate GrayScale image. [This step has already been done using the Otsu thresholding algorithm]
4. **Speckle Noise Removal:** Noise (small dots or foreground components) may be introduced easily into an image while scanning it during Image Acquisition because of various reasons like low clarity camera, a shadow on image etc. This noise should be removed so that the image will be clean and uniform.
5. **Thinning:** Different images have text with different width of strokes. This variability is very high in the case of handwritten words. Skeletonization is a technique, using which we can make all strokes to have a uniform width (Maybe 1 pixel wide or few pixels wide)

3.1 Perspective transform

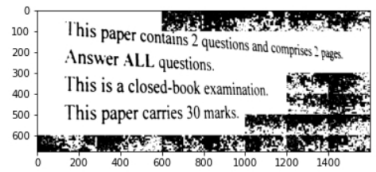
```
[28] # Loading a 3D perspective image
     paper = load_img("/content/paper.jpeg")
     display(paper)
```




```
[29] # Noting down original text for image
     original_text_paper = "This paper contains 2 questions and comprises 2 pages. \n Answer ALL questions. \n This is a closed-book examination. \n This paper carries 30 marks"
```

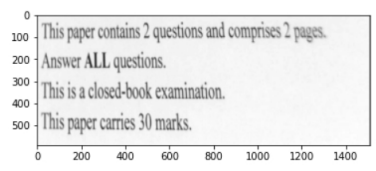
```
[30] # Conducting thresholding and OCR on image
     adaptive_otsu_paper = adaptive_otsu(paper, 100, 200)
     display(adaptive_otsu_paper)

     ocr_adaptive_otsu_paper = pytesseract.image_to_string(adaptive_otsu_paper)
     print(ocr_adaptive_otsu_paper)
     print(f"Accuracy: {accuracy(original_text_paper, ocr_adaptive_otsu_paper)}")
```

```
1 . ey 3 i ; a Seah f
his Paper contains ? questio

Answer ALL questions.
This is a closed-book examination. 3
This paper carries 30 marks. jugpsersimeni

s
gg. =: a
an be =

NS and comprises ? pages,

Accuracy: 0.6712486220795378
```




```
[31] def perspective_transform(img, corners):
       (tl, tr, br, bl) = corners

       # Compute width of new image
       width_top = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2)) # Distance between top right and left
       width_bottom = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2)) # Distance between bottom right and left
       width = max(int(width_top), int(width_bottom)) # Maximum of above 2

       # Compute height of new image
       height_left = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2)) # Distance between left top and bottom
       height_right = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2)) # Distance between right top and bottom
       height = max(int(height_right), int(height_left))

       # Creating points to map corner points to in new image
       corners_new = np.array([
         [0, 0],
         [width - 1, 0],
         [width - 1, height - 1],
         [0, height - 1]], dtype = "float32")

       # Compute perspective transform matrix and transform image
       transform_matrix = cv2.getPerspectiveTransform(corners, corners_new)
       transformed_image = cv2.warpPerspective(img, transform_matrix, (width, height))

       return transformed_image
```

```
[32] paper_corners = np.asarray([(100, 10), (1600, 180), (1600, 550), (100, 600)], dtype = "float32")
     transformed_paper = perspective_transform(paper, paper_corners)
     display(transformed_paper)
```

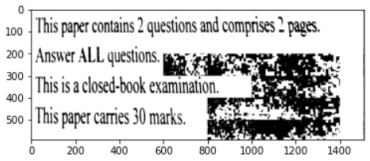



```
[33] # Conducting thresholding and OCR on image
     adaptive_otsu_paper_transformed = adaptive_otsu(transformed_paper, 100, 200)
     display(adaptive_otsu_paper_transformed)

     ocr_adaptive_otsu_paper_transformed = pytesseract.image_to_string(adaptive_otsu_paper_transformed)
     print(ocr_adaptive_otsu_paper_transformed)
     print(f"Accuracy: {accuracy(original_text_paper, ocr_adaptive_otsu_paper_transformed)}")
```

```
This paper contains 2 questions and comprises 2 pages.
Answer ALL questions. era
This is a closed-book examination. |
```

```
This paper carries 30 marks,  &
```

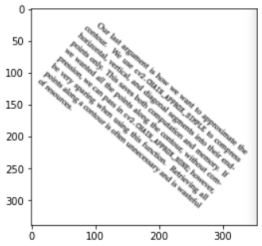
```
Accuracy: 0.8885233166386385
```



Conclusion: As we can see, the image without the transformation was not recognised properly since the OCR algorithm is created for 2D planar text. After we conducted perspective transform on the image, the recognised text is a lot more accurate.

3.2 Image de-skewing

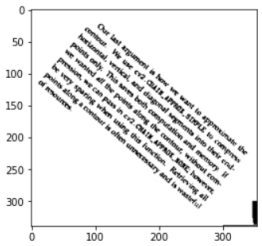
[34]
```
#Loading a 2D-skewed image
skewed_paper = load_img("/content/skewedpaper.png")
display(skewed_paper)
```



[35]
```
# Noting down original text for image
original_text_skewed = "Our last argument is how we want to approximate the contour. We use cv2.CHAIN_APPROX_SIMPLE to compress horizontal, vertical, and diagonal segments into their e
```

[36]
```
# Conducting thresholding and OCR on image
adaptive_otsu_paper_skewed = adaptive_otsu(skewed_paper, 300, 300)
display(adaptive_otsu_paper_skewed)

ocr_adaptive_otsu_paper_skewed = pytesseract.image_to_string(adaptive_otsu_paper_skewed)
print(ocr_adaptive_otsu_paper_skewed)
print(f"Accuracy: {accuracy(original_text_skewed, ocr_adaptive_otsu_paper_skewed)}")
```

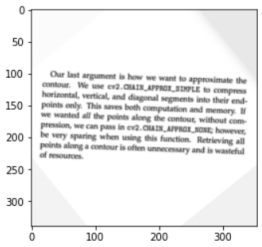
```
Accuracy: 0.0
```



[37]
```
#Defining the deskew functions

# Calculate skew angle of an image
def getSkewAngle(cvImage):

    # Prep image, copy, convert to gray scale, blur, and threshold
    newImage = cvImage.copy()
    blur = cv2.GaussianBlur(newImage, (9, 9), 0)
    thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)[1]

    # Apply dilate to merge text into meaningful lines/paragraphs.
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (30, 5))
    dilate = cv2.dilate(thresh, kernel, iterations=5)

    # Find all contours
    contours, hierarchy = cv2.findContours(dilate, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    contours = sorted(contours, key = cv2.contourArea, reverse = True)

    # Find largest contour and surround in min area box
    largestContour = contours[0]
    minAreaRect = cv2.minAreaRect(largestContour)

    # Determine the angle. Convert it to the value that was originally used to obtain skewed image
    angle = minAreaRect[-1]
    if angle < -45:
        angle = 90 + angle
    return -1.0 * angle


# Rotate the image around its center
def rotateImage(cvImage, angle: float):
    newImage = cvImage.copy()
    (h, w) = newImage.shape[:2]
    center = (w // 2, h // 2)
```

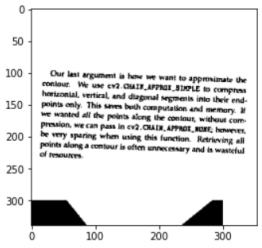
```
        M = cv2.getRotationMatrix2D(center, angle, 1.0)
        newImage = cv2.warpAffine(newImage, M, (w, h), flags=cv2.INTER_CUBIC, borderMode=cv2.BORDER_REPLICATE)
        return newImage

    # Deskew image
    def deskew(cvImage):
        angle = getSkewAngle(cvImage)
        return rotateImage(cvImage, -1.0 * angle)
```

```
[38] deskewed_paper = deskew(skewed_paper)
     display(deskewed_paper)
```



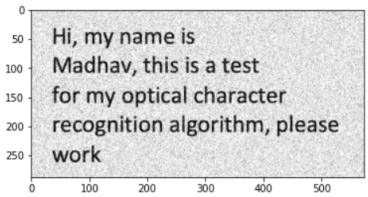

```
[39] # Conducting thresholding and OCR on image
     adaptive_otsu_paper_deskewed = adaptive_otsu(deskewed_paper, 300, 300)
     display(adaptive_otsu_paper_deskewed)

     ocr_adaptive_otsu_paper_deskewed = pytesseract.image_to_string(adaptive_otsu_paper_deskewed)
     print(ocr_adaptive_otsu_paper_deskewed)
     print(f"Accuracy: {accuracy(original_text_skewed, ocr_adaptive_otsu_paper_deskewed)}")
```

```
Dur last argument is how we want to approslinate the
fonlaus. We use c¥2.CHAIH_APPRUX,SINPLE to compnss
harizonial, vertical, and diagonal segments into their end
Points only. This saves both computation andl memory. It
we wanted ell the points slong the conlaut, without com:
Prrasion, we can pass in cv2. CHAIR. APPROZ_ ROME; howeves,
be very sparing when using this function. Retrieving all
Points along a contour is often unnecessary and is wasteful
of resources.

—_ @a

Accuracy: 0.6378635118111463
```



3.4 Removing Speckle Noise

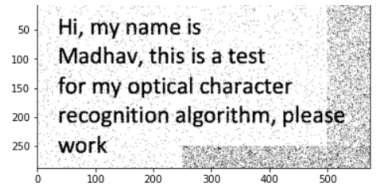
```
[40] #Loading a speckled image
     speckle = load_img("/content/noise2.png")
     display(speckle)
```



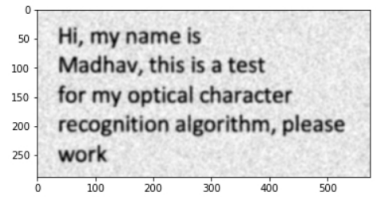

```
[41] # Noting down original text for image
     original_text_noise = "Hi, my name is Madhav, this is a test for my optical character recognition algorithm; please work"
```

```
[42] # Conducting thresholding and OCR on image
     adaptive_otsu_speckle = adaptive_otsu(speckle, 250, 250)
     display(adaptive_otsu_speckle)

     ocr_adaptive_otsu_speckle = pytesseract.image_to_string(adaptive_otsu_speckle)
     print(ocr_adaptive_otsu_speckle)
     print(f"Accuracy: {accuracy(original_text_noise, ocr_adaptive_otsu_speckle)}")
```

```
Hi, myname is

~ Madhary, this is a test
for my optical character

“recognition algorithm, pleas
work eee




Accuracy: 0.6172133998483676
```

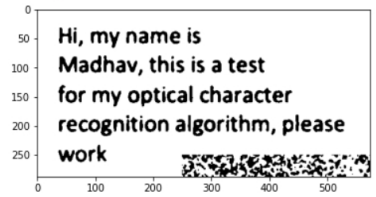


[43]
```
#Removing noise using cv2 gaussian filter
gaussian_speckle = cv2.GaussianBlur(speckle, (9, 9), 0)

# Plotting of image after Gaussian Filter
display(gaussian_speckle)
plt.show()
```



[44]
```
# Conducting thresholding and OCR on image
adaptive_otsu_gaussian = adaptive_otsu(gaussian_speckle, 250, 250)
display(adaptive_otsu_gaussian)

ocr_adaptive_otsu_gaussian = pytesseract.image_to_string(adaptive_otsu_gaussian)
print(ocr_adaptive_otsu_gaussian)
print(f"Accuracy: {accuracy(original_text_noise, ocr_adaptive_otsu_gaussian)}")
```

```
Hi, my name is

Madhav, this is a test

for my optical character
recognition algorithm, please

work FRYE IWR

Accuracy: 0.8807048459279793
```



As it can be clearly seen, the output after removing noise using the Gaussian filter is much clearer than without removing it.

3.5 Thinning

Thinning in OCR is required mostlly with handwritten documents, when the text is too thick and thus some characters overlap with the others. Thinning reduces the width of each character, thus reducing the overlap and making it easier for the OCR algorithm to detect indivudual characters

[45]
```
#Loading a speckled image
comp1 = load_img("/content/cz40032.png")
# comp1 = cv2.rotate(comp1, cv2.ROTATE_90_COUNTERCLOCKWISE)
plt.imshow(comp1, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7fa284652f60>
```

[46]
```
# Noting down original text for image
original_text_thin = "CZ 4003"
```

[47]
```
# Conducting thresholding and OCR on image
adaptive_otsu_thin = adaptive_otsu(comp1, 50, 50)
display(adaptive_otsu_thin)

ocr_adaptive_otsu_thin = pytesseract.image_to_string(adaptive_otsu_thin)
print(ocr_adaptive_otsu_thin)
print(f"Accuracy: {accuracy(original_text_thin, ocr_adaptive_otsu_thin)}")
```

```
CZ 4003

Accuracy: 0.5
```

[48]
```
kernel = np.ones((7,7),np.uint8)
thinned_comp1 = cv2.erode(comp1, kernel,iterations = 1)

display(thinned_comp1)
```

[49]
```
# Conducting thresholding and OCR on image
adaptive_otsu_thinned = adaptive_otsu(thinned_comp1, 50, 50)
display(adaptive_otsu_thinned)

ocr_adaptive_otsu_thinned = pytesseract.image_to_string(adaptive_otsu_thinned)
print(ocr_adaptive_otsu_thinned)
print(f"Accuracy: {accuracy(original_text_thin, ocr_adaptive_otsu_thinned)}")
```

```
CZ 4003

Accuracy: 1.0
```