TEXT EXTRACTION OF NATURAL IMAGES

Submitted by

R.Anuraag (16BCE0752)

Akshay Arora (16BCE0944)

CSE3019

Image Processing

Slot: G2

In partial fulfilment for the award of the degree of

B.Tech

in

Computer Science and Engineering

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NOVEMBER, 2018

Text Extraction and Recognition from Natural Images

R.Anuraag - 16BCE0752 Akshay Arora- 16BCE0944

***Abstract* -Image based text extraction is one of the fastest growing research areas in the field of multimedia technology. Text detection in natural scenes is an important but challenging problem because of variations in the text fonts, size, line orientation, and complex background in image and non-uniform illuminations. Also, the extraction of text from a complex or more colorful images is a challenging problem. Text data present in images contains useful information for habitual explanation, indexing, and structuring of images. The major processes to implement text extraction can be classified as Text Preprocessing, Detection and Text Recognition. [1] There are several algorithms provided to complete these two processes with maximum accuracy.**
Our plan is to implement accurately text extraction and recognition using all major algorithms which are defined for the above mentioned processes using Python and OpenCV.

Keywords: MSER (Maximally Stable Extremal Regions), Contours, Text Detection, Text Extraction

1. INTRODUCTION

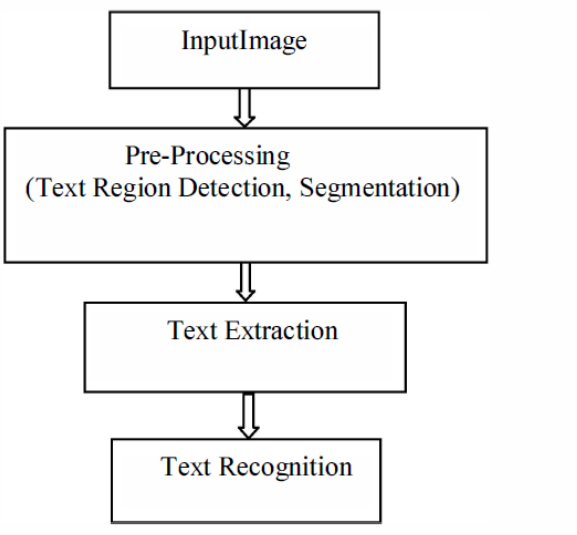
Text Detection and Extraction of natural scenes can be defined as detecting and segmenting the text regions in an otherwise image oriented image. This is currently considered to be an important and challenging task as stated above, the reason being the different orientation of images and different font. [1]
We read several research papers to find algorithms that provide us efficient results.

We found that OpenCV (An open source image processing software is available and it has several default filters that can be helpful to provide us with satisfactory outputs). So, we are currently using Python + OpenCV for our text detection and extraction process.

But first, we needed to know about the procedure that needs to be followed to process the image as well as to read about the new and necessary algorithms to be used.

The working procedure for text extraction/detection is as follows: [2]

1. Pre- Processing
2. Text Localization or Segmentation
3. Implementation of Text Localization Algorithms



Explanation about each step for text detection:

1. Pre-Processing

The text extraction process starts with the very important pre-processing phase. It includes all processes starting with conversion of the color image to gray scale image to smoothening and thresholding of the image.

2. Text Localization

The objective of text localizer is to highlight the letters in an image, using either a rectangle, or marking the area around a particular letter in all the words present in the image, can be considered as an option. The output of text detector is a boundary –box covering text regions and some non-text regions.

2. OBJECTIVE

i) Our objective is to input a coloured image and extract and recognize the text using sequence of steps performed using Python and OpenCV
ii) Implement properly all the different text detection algorithms.
iii) Detect text from image of different types (White Text over coloured, Coloured Text, or Simple text on a plain Background) using proper Pre-processing methods.
iv) Next objective, is to compare all the methods of text detection, and use the best one for text extraction of images with similar backgrounds.

3. RELATED WORKS

Researched about the approaches that are present to the processes mentioned above.

Approach for Text Localization
There are several methods of text localizing, and we have tried to implement all of them to our best extent.

The methods/algorithms are:-

Algorithms for Text Localization:

i) Maximally Stable Extremal Regions (MSER) Detection[1]
MSER is a method for blob detection in images.The job of the MSER is to extract many regions that are co-variant to each other. A MSER is a stable connected component of some gray level sets of the image.
MSER is based on the idea of taking regions which stay nearly the same through a wide range of thresholds.

ii) Edge Detection:
This algorithm is called Canny Edge Detection, as it states it is a multistep algorithm used to remove noise and bring out edges. Contour detection is an important part of edge detection too

iii) Contour Detection:
As the name suggests, this is used to detect and mark the boundary of an object.

3. Text Extraction and Recognition[6]

Algorithms for text recognition

1. First locating text regions, then merging all the text regions as one and use the OCR (optical character recognition function to read it).
2. Take all the text regions, segment them into letters and then use a training set of letters to recognize the text.
3. Use of SVM (Support Vector Machines) and use a big training data set. - Useful when needed to recognize offline handwriting habits.

For, this project, this job is done by pytesseract which a default python OCR.

4. MOTIVATION

When researching about Image Processing, the most sought after field was that of text extraction and recognition. The reason being text extraction is a field that is still under research and no particular solution is still found for the same.

There are certain filters present, but no generic solution has still been discovered. As said, each image has its own properties, and to extract it might require a unique combination of filters working together.

The newness and the challenging component is what drew us towards Text Extraction.

Also, in this time, when Machine Learning and AI is so prominent, Text Extraction has become really important and an essential factor of automating any available process.

5. FUNCTIONAL REQUIREMENTS

The following project requires the following softwares to function:

1. Python IDE and its libraries
2. Open CV working with Python
3. MSER and Canny Edge Detection in openCV
4. Addition of numpy and matplotlib libraries in Python
5. Presence and allowance of giving attributes to function class.

Working Requirement

This is basic but essential knowledge required to run this project.

1. Knowledge about various python libraries
2. Working of various filters in image processing
    - Thresholding
    - Opening
    - Closing
    - Erosion
    - Dilation
    - Binarization

6. METHODOLOGY

We can divide our working to cater to three different types of images

1. Black Text with Plain Background
2. White text with image background
3. Similar(RGB) colours in the background

Each of these images have a different methodology that needs to be followed or implemented

1st Method (For a Plain Background image) [4]

Working

Step 1: Pre-Processing of image
Resizing the image acc. to the original image
Converting to Grayscale
Perform Binary Threshold
Apply a mask of binary Threshold on Grayscale image
Use iterative dilation to dilate more.

Step 2: Text Localization done by all different algorithms.

These are implemented using several different inbuilt functions in Opencv

- Contour Detection
- MSER Detection: Uses Convex Hull and further uses contour marking to get output
- Edge Detection: After further smoothening of image using thresholding, we use canny command to get edges of the text.

The smoothening process in this method constitutes multiple thresholds for noise removal and use of MedianBlur which is non-linear in nature.

Step 3: Further Image Pre-processing for OCR, After Choosing Edge Detection as primary text localization algorithm.

- Adaptive Thresholding
- Morphological Operations like opening and closing, are used for noise removal as well as removal of holes, are used for differentiating between the edge and the text.

Step 4: Printing the text

Use of pytesseract, we read the text in the image, then we use file handling, to send the text to a .txt file to store.

The pytesseract is a default OCR provided by Python, it is used by downloading an .exe file

2nd Method (For a White text image on an image as a background)

Step 1: Pre-Processing of image
Resizing the image acc. to the original image
Converting to Grayscale
Perform Binary Threshold
Apply a mask of binary Threshold on Grayscale image
Use iterative dilation to dilate more.
Requires further binary inverse thresholding.

Step 2: Text Localization done by all different algorithms.

These are implemented using several different inbuilt functions in Opencv

- Contour Detection
- MSER Detection: Uses Convex Hull and further uses contour marking to get output
- Edge Detection: After further smoothening of image using thresholding, we use canny command to get edges of the text.

The smoothening process in this method constitutes multiple thresholds for noise removal and use of **GaussianBlur** which is linear in nature, used for noise removal as well as reducing contrast.

Step 3: Further Image Pre-processing for OCR, After Choosing Edge Detection as primary text localization algorithm.

- Adaptive Thresholding
- Morphological Operations like opening and closing, are used for noise removal as well as removal of holes, are used for differentiating between the edge and the text.
- **Use a binary inversion threshold to get the text black, to make it easier for the OCR to read. This is a differential point in both the images.**

Step 4: Printing the text

Use of pytesseract, we read the text in the image, then we use file handling, to send the text to a .txt file to store.

The pytesseeact is a default OCR provided by Python, it is used by downloading an .exe file

3nd Method (similar background and foreground) [3]

The final steps remain the same only the pre-processing changes

Step 1: Pre-Processing

Converting the image into hsv to add a different overlay

Binarizing the hsv image to product a distinct image and then further smoothening

Now further smoothening the same using repeated dilation until we get a clear distinction.

Then attempt edge detection and then further thresholding, to make the distinction.

Step 2: Print using pytesseract

7. CONCEPTS USED[2]

I. Thresholding

Thresholding is the simplest method of image segmentation. It can create an array of binary images, ranging from a grayscale image, to a binary reverse to a binary null image.

There are various types of thresholding present:

1. Global Thresholding

   It is used to create a division between two peaks, now, it uses multiple threshold iterations, and by continuously changing the threshold values acc. to the existing mean fo the colour code values.

   It is essential in this code, to make image binary to an accurate extent and vice-versa.

2. Adaptive Thresholding

   - The use of mean/median of an image as a pre-existing value.
   - Subtracting it from the original image.
   - And, subsequently thresholding, to get a binary image.
   - Essential pre-processing part for distinction between edges and surroundings.

3. Otsu's Thresholding

   The algorithm assumes that the image contains two foreground pixels and background pixels, it then calculates the optimum threshold separating the two classes so that their combined spread variance is minimal.

Syntax: ret2, th2 = cv2.threshold (th1, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU) [5]

II. Morphological Operations

Five morphological operations have been the essence and the main reason for this project to work.

i. Erosion
ii. Dilation
iii. Opening
iv. Closing

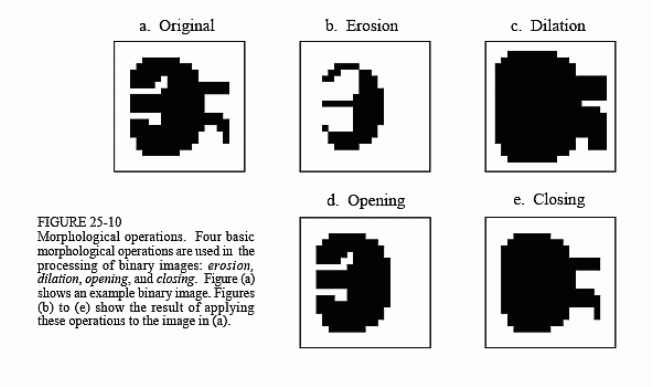


FIGURE 25-10
Morphological operations. Four basic morphological operations are used in the processing of binary images: *erosion*, *dilation*, *opening*, and *closing*. Figure (a) shows an example binary image. Figures (b) to (e) show the result of applying these operations to the image in (a).

Concept of Structuring Element (kernel)

A kernel is essential for comparison ad taking action in all the morphological operations.

The **structuring element** can be defined a small binary image, like a matrix of pixels, each with a value of zero or one:

- The matrix dimensions specify the *size* of the structuring element.
- The pattern of ones and zeros specifies the *shape* of the structuring element.

i. Erosion
It can be defined as shrinking away of pixels around the edge as well as removal of holes

ii. Dilation
It is the addition of extra pixel around the edges, as well as proper filling of holes.

More important operations are opening and closing, these have been used abundant in the code, for edge processing, preferred due to their noise removing properties.

iii. Opening(Erosion +Dilation)

Opening is so called used for opening up gaps between objects connected by a thin bridge of pixels. Any regions that have undergone the erosion and still remain are restored to their original size by the dilation

Syntax: [5]
kernel = np.ones ((1, 1), np.uint8)
opening = cv2.morphologyEx(filtered, cv2.MORPH_OPEN, kernel)

iv. Closing(Dilation + Erosion)
This is basically the opposite of opening, it is used to fill up holes.

Syntax:[5]

closing=cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel)

Note:

In the code, Opening is done followed by closing. Closing in done on the opening output, just to fill up any space discrepancies, plus further noise removal.

Followed by a masking operation.

III. Importance of Resizing

Resizing turned out to be one of the most important factors, with respect to providing accurate output.

Syntax:

image = cv2.resize(image1, None, fx=x1, fy=y1, interpolation=cv2.INTER_CUBIC)

In the syntax, fx and fy denote the factors by which size changes(shrunk or enlarge). CUBIC – is an accurate but slow resizing method to use.

Resizing an image by (fx,fy) factors, helps the image to be the right size on the screen for the OCR to read. Either size extermities in this case could cause a big problems.

To solve this, we implemented an if function to monitor the height and width of the image.

Solution syntax:

```
def x_values(h,w):
    if h<=200 and w<=300:
        x=2.5
    elif h<=500 and w<600:
        x=2.0
    elif h<=600 and w<=800:
        x=1.4
    else:
```

```
        x=1.2
    return x
def y_values(h,w):
    if h<=200 and w<=300:
        y=2.5
    elif h<=500 and w<600:
        y=2.0
    elif h<=600 and w<=800:
        y=1.4
    else:
        y=1.2
    return y
```

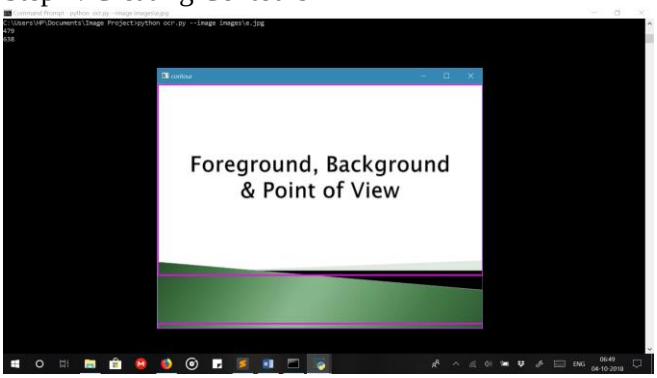
Getting, the values of fx and fy from this if function, improved the accuracy of the code by a whopping 75% margin

8. OUTPUTS AND OBJECTIVES

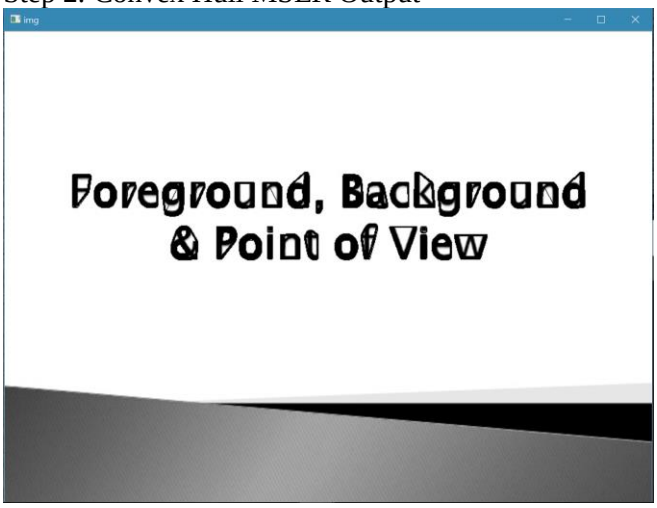
1. A simple image(Plain Background)

I. Black Text on a White Background

Step 1: Creating Contours

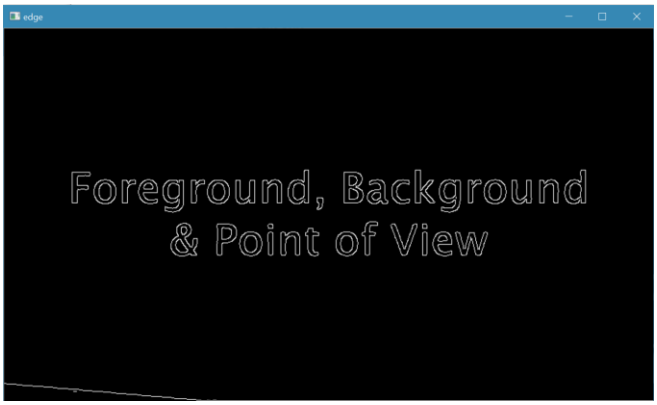


Step 2: Convex Hull MSER Output

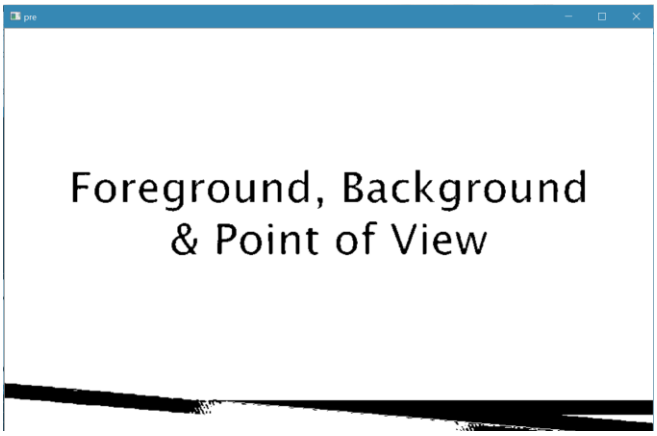


Step 3: MSER Detected Image

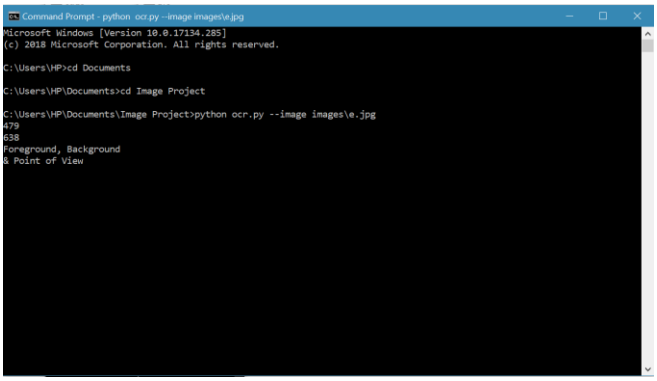
Step 4: Canny Edge Detection



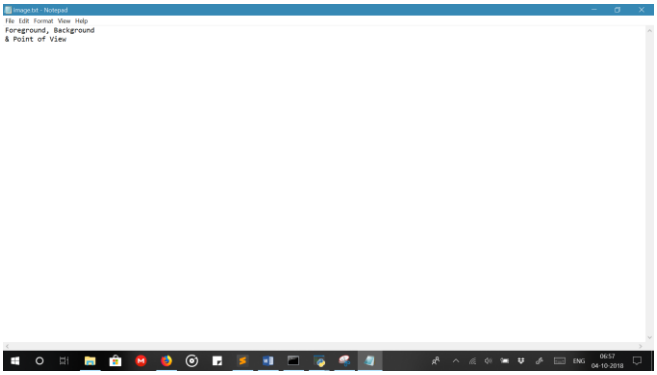
Step 5 :Processing result



Step 6: Output



Step 7: Writing in a File (image.txt)



2. A white text on a darker background(Complex Image)

=> 3rd Objective Completed and Implemented

Steps to text extraction:

Step 1 :Contour marking

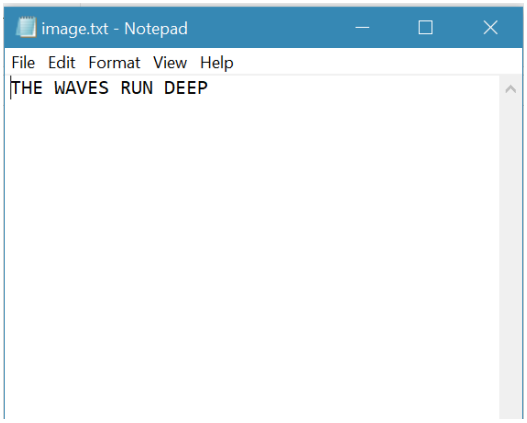
Step 2:Text_Only(MSER)

Step 3:Edge Detection

Step 4: Opening/Closing (Processing)

Step 5: Binary Inversion of earlier Image

Step 6: Text Extracted

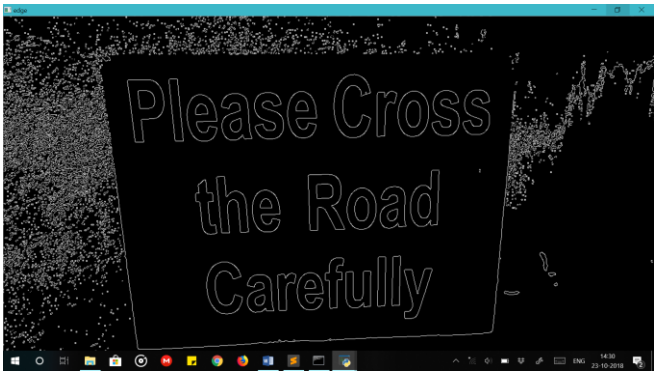


9. LIVE APPLICATION

We chose to use this text extraction on a sign present on out campus.

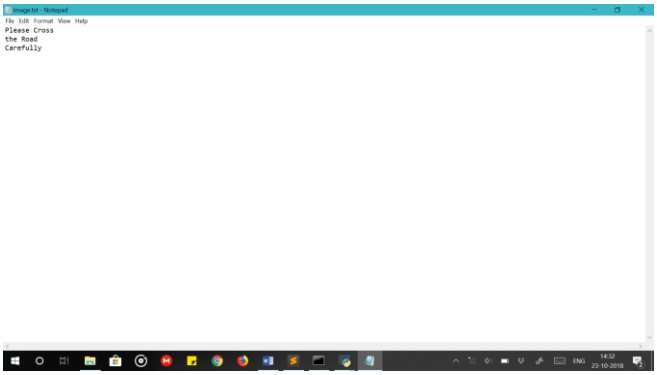
Figure: Natural Image (From VIT Campus)

Step 1: MSER followed by Canny Edge detection



Step 2: Thresholding and Binarization

Step 3: Get the output

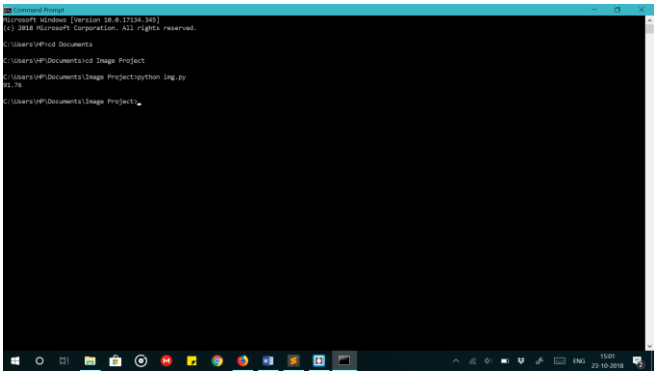


10. ADDITIONAL COMPONENTS

We were asked to implement handwriting recognition, if possible, so as to extract digits or images from the same.

We tried to do the same, but the output wasn't as perfect as we intended.
But, we are able to extract digits.png from OpenCV and apply KNN on it.

For Code refer, Appendix II



11. CONCLUSION

We have completed the all the objectives of our project as shown in the results above.
The final objective, was to complete the implementation of similar background and foreground text images before the final review, which is completed.
Another additional component of handwriting extraction was added, this was also attempted.

12. REFERENCES

[1] ROBUST TEXT DETECTION IN NATURAL IMAGES WITHEDGE-ENHANCED MAXIMALLY STABLE EXTREMAL REGIONS
Huizhong Chen, Sam S. Tsai, Georg Schroth, David M. Chen, Radek Grzeszczuk and Bernd Girod

[2] An Efficient Algorithm for Text Localization and Extraction in Complex Video Text Images
Anubhav Kumar , Member, IEEE ,Neeta Awasthi Assistant Professor, Director

[3] 17th International Conference on Computer and Information Technology (ICCIT)
Connected Component Based Approach for Text Extraction from Color Image

[4] Integrated Natural Scene Text Localization and Recognition
Kakade Snehal Satwashil1, Prof.Dr.V.R.Pawar2
Department of Electronics and Telecommunication
University of Savitribai Phule, BVCOEW, Dhankwadi

[5] https://opencv.org/

[6] TEXT LOCALIZATION AND EXTRACTION IN IMAGES USING MATHEMATICAL MORPHOLOGY AND SVM
R.Chandrasekaran, RM.Chandrasekaran, P.Natarajan
Associate Professor, Dept. of Computer Science & Engg., Annamalai University, India

[7] International Conference on Communication and Signal Processing
Text Detection and Recognition in Natural Scene Images
Amruta Pise, S.D.Ruikar

13. APPENDIX

Code for the above outputs using python and open cv

```
# import the necessary packages
from PIL import Image
import pytesseract
import argparse
import cv2
import os
import numpy as np
# from matplotlib import pyplot as plt
pytesseract.pytesseract.tesseract_cmd = r"C:\\Program Files (x86)\\Tesseract-OCR\\tesseract.exe"

def image_smooth(img):
        ret1, th1 = cv2.threshold(img,127, 255, cv2.THRESH_BINARY)
        ret2, th2 = cv2.threshold(th1, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        blur = cv2.medianBlur(th2,3)
        ret3, th3 = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        return th3

def image_smooth2(img):
        ret1, th1 = cv2.threshold(img,180, 255, cv2.THRESH_BINARY)
        ret2, th2 = cv2.threshold(th1, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        blur = cv2.GaussianBlur(th2,(1, 1), 0)
        ret3, th3 = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        return th3

def mser(image):
        #Detecting Text Regions
        #Create MSER object
        mser = cv2.MSER_create()

        vis = image.copy()

        regions, _ = mser.detectRegions(image)
        #Use to extend the text regions to extend over sharp turns
        hulls = [cv2.convexHull(p.reshape(-1, 1, 2)) for p in regions]

        cv2.polylines(vis, hulls, 1, (0, 255, 0))

        cv2.imshow('img', vis)

        mask = np.zeros((image.shape[0], image.shape[1], 1), dtype=np.uint8)

        for contour in hulls:
                cv2.drawContours(mask, [contour], -1, (255, 255, 255), -1)

        #this is used to find only text regions, remaining are ignored
        text_only = cv2.bitwise_and(image, image, mask=mask)

        cv2.imshow("text only", text_only)
        cv2.waitKey(0)
        return text_only

# def contours(img):

#       gray1= cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#       ret6, mask = cv2.threshold(gray1, 180, 255, cv2.THRESH_BINARY)
#       image_final = cv2.bitwise_and(gray1, gray1, mask=mask)
```

```
#           ret6, new_img = cv2.threshold(image_final, 180,
255, cv2.THRESH_BINARY)
#           kernel =
cv2.getStructuringElement(cv2.MORPH_CROSS, (3,
#
                                    3))
#           # to manipulate the orientation of dilution ,
#           dil = cv2.dilate(new_img, kernel, iterations=5)
#           #Adding Contours
#           _, contours, hierarchy = cv2.findContours(dil,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
#           for contour in contours:
#                   [x, y, w, h] = cv2.boundingRect(contour)
#                   # draw rectangle around contour on
original image
#                   if w < 35 and h < 35:
#                           pass
#                   cv2.rectangle(img, (x, y), (x + w, y + h),
(255, 0, 255), 2)
#                   cropped = image_final[y :y +  h , x : x +
w]
#                   cv2.imshow('contours',img)
#                   cv2.waitKey(0)
#                   cv2.imshow('cropped',img)
#                   cv2.waitKey(0)
#                   return img


def file1(txt):
        #Writing data in the file
        f=open("C:\\Users\\HP\\Documents\\Image
Project\\image.txt","w+")
        f.write(txt)
        f.close()
        return 0

def x_values(h,w):
        if h<=200 and w<=300:
                x=2.5
        elif h<=500 and w<600:
                x=2.0
        elif h<=600 and w<=800:
                x=1.4
        else:
                x=1
        return x


def y_values(h,w):
        if h<=200 and w<=300:
                y=2.5
        elif h<=500 and w<600:
                y=2.0
        elif h<=600 and w<=800:
                y=1.4
        else:
                y=1
        return y


ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
        help="specify path to input image")
ap.add_argument("-p", "--preprocess", type=str,
default="thresh",
        help="type of preprocessing to be done")
args = vars(ap.parse_args())

#Resize Parameters
image1 = cv2.imread(args["image"])
im = Image.open(args["image"])
width, height = im.size
print(height)
print(width)

x1=x_values(height,width)
y1=y_values(height,width)


image = cv2.resize(image1, None, fx=x1, fy=y1,
interpolation=cv2.INTER_CUBIC)

gray= cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

if args["preprocess"] == "text":
        #Contours
        gray1= cv2.cvtColor(image1,
cv2.COLOR_BGR2GRAY)
        ret, mask = cv2.threshold(gray1, 180, 255,
cv2.THRESH_BINARY)
        image_final = cv2.bitwise_and(gray1, gray1,
mask=mask)
        ret, new_img = cv2.threshold(image_final, 180,
255, cv2.THRESH_BINARY)  # for black text ,
cv.THRESH_BINARY_INV
        kernel =
cv2.getStructuringElement(cv2.MORPH_CROSS, (3,


                                3))  # to manipulate the
orientation of dilution , large x means horizonatally dilating
more, large y means vertically dilating more
        dilated = cv2.dilate(new_img, kernel, iterations=9)
        # dilate , more the iteration more the dilation
        _, contours, hierarchy = cv2.findContours(dilated,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
        # findContours returns 3 variables for getting
contours

        for contour in contours:
                [x, y, w, h] = cv2.boundingRect(contour)

                # Don't plot small false positives that
aren't text
                if w < 35 and h < 35:
                        continue
                # draw rectangle around contour on
original image
                cv2.rectangle(image1, (x, y), (x + w, y +
h), (255, 0, 255), 2)
        cv2.imshow('contour',image1)
        cv2.waitKey(0)

        #mser
        img2 = mser(gray)
        #inverting the grayscale image
        new=image_smooth2(gray)
        ret4, th4 = cv2.threshold(new,155, 255,
cv2.THRESH_BINARY)
        #Canny Edge Detection
        edges = cv2.Canny(th4,100,200)
```

```
        cv2.imshow('edge',edges)
        cv2.waitKey(0)

        #Pre-Processing
        filtered =
cv2.adaptiveThreshold(edges.astype(np.uint8), 255,
cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY, 41, 3)
        kernel = np.ones((1, 1), np.uint8)
        opening = cv2.morphologyEx(filtered,
cv2.MORPH_OPEN, kernel)
        closing = cv2.morphologyEx(opening,
cv2.MORPH_CLOSE, kernel)
        or_image = cv2.bitwise_or(th4, closing)
        ret5, th5 = cv2.threshold(or_image,140, 255,
cv2.THRESH_BINARY)
        cv2.imshow('pre',or_image)
        cv2.waitKey(0)
        cv2.imshow('next',th5)

        #Print the filenames
        filename = "{}.png".format(os.getpid())
        cv2.imwrite(filename,th5)

        text =
pytesseract.image_to_string(Image.open(filename))
        os.remove(filename)
        print(text)
        file1(text)
        cv2.waitKey(0)


elif args["preprocess"]=="thresh":
        #Contours
        gray1= cv2.cvtColor(image1,
cv2.COLOR_BGR2GRAY)
        ret, mask = cv2.threshold(gray1, 180, 255,
cv2.THRESH_BINARY)
        image_final = cv2.bitwise_and(gray1, gray1,
mask=mask)
        ret, new_img = cv2.threshold(image_final, 180,
255, cv2.THRESH_BINARY)  # for black text ,
cv.THRESH_BINARY_INV
        kernel =
cv2.getStructuringElement(cv2.MORPH_CROSS, (3,


                                3))  # to manipulate the
orientation of dilution , large x means horizonatally dilating
more, large y means vertically dilating more
        dilated = cv2.dilate(new_img, kernel, iterations=9)
        # dilate , more the iteration more the dilation
        _, contours, hierarchy = cv2.findContours(dilated,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
        # findContours returns 3 variables for getting
contours

        for contour in contours:
                [x, y, w, h] = cv2.boundingRect(contour)

                # Don't plot small false positives that
aren't text
                if w < 35 and h < 35:
                        continue
                # draw rectangle around contour on
original image
                        cv2.rectangle(image1, (x, y), (x + w, y +
h), (255, 0, 255), 2)
        cv2.imshow('contour',image1)
        cv2.waitKey(0)

        #mser
        img2= mser(gray)
        #inverting the grayscale image
        #Canny Edge Detection
        edges = cv2.Canny(gray,100,200)
        cv2.imshow('edge',edges)
        cv2.waitKey(0)

        #Pre-Processing
        filtered =
cv2.adaptiveThreshold(edges.astype(np.uint8), 255,
cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY, 41, 3)
        kernel = np.ones((1, 1), np.uint8)
        opening = cv2.morphologyEx(filtered,
cv2.MORPH_OPEN, kernel)
        closing = cv2.morphologyEx(opening,
cv2.MORPH_CLOSE, kernel)
        new1=image_smooth(gray)
        or_image = cv2.bitwise_or(new1, closing)
        cv2.imshow('pre',or_image)

        filename = "{}.png".format(os.getpid())
        cv2.imwrite(filename, or_image)

        text =
pytesseract.image_to_string(Image.open(filename))
        os.remove(filename)
        print(text)
        file1(text)
        cv2.waitKey(0)

elif args["preprocess"]=="option":

        ret, gray = cv2.threshold(gray, 230, 255,
cv2.THRESH_BINARY | cv2.THRESH_OTSU)
        img2= mser(gray)
        #hsv= cv2.cvtColor(image1,
cv2.COLOR_BGR2HSV)
        #Canny Edge Detection
        cv2.imshow('mser',img2)
        cv2.waitKey(0)
        #Preprocessing for colored backgrounds
        '''
        hsv= cv2.cvtColor(hsv,
cv2.COLOR_BGR2GRAY)
        cv2.imshow('hsv',hsv)
        cv2.waitKey(0)
        '''
        filtered =
cv2.adaptiveThreshold(img2.astype(np.uint8), 255,
cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY, 41, 3)
        kernel = np.ones((1, 1), np.uint8)
        opening = cv2.morphologyEx(filtered,
cv2.MORPH_OPEN, kernel)
        closing = cv2.morphologyEx(opening,
cv2.MORPH_CLOSE, kernel)
        new1=image_smooth(gray)
        or_image = cv2.bitwise_or(new1, closing)
        cv2.imshow('pre',or_image)
```

```
        filename = "{}.png".format(os.getpid())
        cv2.imwrite(filename, or_image)

        text =
pytesseract.image_to_string(Image.open(filename))
        os.remove(filename)
        print(text)
        file1(text)
        cv2.waitKey(0)
```

APPENDIX II

```
import cv2
from sklearn.externals import joblib
from skimage.feature import hog
import numpy as np
import argparse as ap

# Get the path of the training set
pars = ap.ArgumentParser()
pars.add_argument("-c", "--classiferPath", help="Path",
required="True")
pars.add_argument("-i", "--image", help="Path to Image",
required="True")
args = vars(pars.parse_args())

# Load the classifier
clf, pp = joblib.load(args["classiferPath"])

# Read the input image
im = cv2.imread(args["image"])

# Convert to grayscale and apply Gaussian filtering
im_gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
im_gray = cv2.GaussianBlur(im_gray, (5, 5), 0)

# Threshold the image
ret, im_th = cv2.threshold(im_gray, 90, 255,
cv2.THRESH_BINARY_INV)

# Find contours in the image
_, ctrs, hier = cv2.findContours(im_th.copy(),
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Get rectangles contains each contour
rects = [cv2.boundingRect(ctr) for ctr in ctrs]

# For each rectangular region, calculate HOG features and
predict
# the digit using Linear SVM.
for rect in rects:
    # Draw the rectangles
    cv2.rectangle(im, (rect[0], rect[1]), (rect[0] + rect[2],
rect[1] + rect[3]), (0, 255, 0), 3)
    # Make the rectangular region around the digit
    leng = int(rect[3] * 1.6)
    pt1 = int(rect[1] + rect[3] // 2 - leng // 2)
    pt2 = int(rect[0] + rect[2] // 2 - leng // 2)
    roi = im_th[pt1:pt1+leng, pt2:pt2+leng]
    # Resize the image
    roi = cv2.resize(roi, (28, 28),
interpolation=cv2.INTER_AREA)
    roi = cv2.dilate(roi, (3, 3))
    # Calculate the HOG features
    roi_hog_fd = hog(roi, orientations=9,
pixels_per_cell=(14, 14), cells_per_block=(1, 1),
visualise=False)
    roi_hog_fd = pp.transform(np.array([roi_hog_fd],
'float64'))
    nbr = clf.predict(roi_hog_fd)
    cv2.putText(im, str(int(nbr[0])), (rect[0],
rect[1]),cv2.FONT_HERSHEY_DUPLEX, 2, (0, 255, 255),
3)

cv2.namedWindow("Resulting Image with Rectangular
ROIs", cv2.WINDOW_NORMAL)
cv2.imshow("Resulting Image with Rectangular ROIs", im)
cv2.waitKey()
```

Using KNN

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
img = cv.imread('digits.png')
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
# Now we split the image to 5000 cells, each 20x20 size
cells = [np.hsplit(row,100) for row in np.vsplit(gray,50)]
# Make it into a Numpy array. It size will be (50,100,20,20)
x = np.array(cells)
# Now we prepare train_data and test_data.
train = x[:,:50].reshape(-1,400).astype(np.float32) # Size =
(2500,400)
test = x[:,50:100].reshape(-1,400).astype(np.float32) # Size
= (2500,400)
# Create labels for train and test data
k = np.arange(10)
train_labels = np.repeat(k,250)[:,np.newaxis]
test_labels = train_labels.copy()
# Initiate kNN, train the data, then test it with test data for
k=1
knn = cv.ml.KNearest_create()
knn.train(train, cv.ml.ROW_SAMPLE, train_labels)
ret,result,neighbours,dist = knn.findNearest(test,k=5)
# Now we check the accuracy of classification
# For that, compare the result with test_labels and check
which are wrong
matches = result==test_labels
image = cv.resize(ret, None, fx=100, fy=100,
interpolation=cv.INTER_CUBIC)
cv.imshow('pre',image)
cv.waitKey(0)
correct = np.count_nonzero(matches)
accuracy = correct*100.0/result.size
print( accuracy )
```