CSE483: Computer Vision

Major Task-Phase I

Team members

Name	ID
Anas Salah Abdelrazik SaadEldin	19P9033
Alaa Mohamed Mohamed Hamdy Ahmed	19P6621
Ahmed Amr Mohyeldin Algayar	19P8349

Table of Contents

Table of Figures

In phase 1 we need to localize digits in the images.

Challenges Faced:

1- Digits in the images have different sizes
2- The digits reside on different backgrounds
3- The digits might reside on multiple backgrounds
4- Digits appear in images differently (different representations for each digit)
5- Variant illumination in the images
6- Some images are blurred, and digits are not perfectly visible even to the human eye.
7- Some digits are occluded.
8- Variant number of digits in each image.
9- Digits appear with different colors in the images.
10- Some images are noisy
11- Variant spaces between digits in images (small/large spacing)

Since we are implementing a global solution with traditional computer vision techniques its difficult to find one technique or algorithm that will optimize the results for all the images, we implemented more than 1 algorithm with the calculated the accuracy for each one.

The dataset comes with a matlab file that has the bounding box information stored in **digitstruct.mat** instead of drawn directly on the images in the dataset. The digitstruct.mat file contains a struct called **digitstruct** with the same length as the number of original images. Each element in digitstruct has the following fields: **name** which is a string containing the filename of the corresponding image. **Bbox** which is a struct array that contains the position, size and label of each digit bounding box in the image. In order to be able to compare our algorithms results with the original correct bounding boxes we converted the matlab file into a json file that has the following structure:


```
[
 {
  "filename": "1.png",
  "boxes": [
   {
    "height": 219.0,
    "label": 1.0,
    "left": 246.0,
    "top": 77.0,
    "width": 81.0
   },
   {
    "height": 219.0,
    "label": 9.0,
    "left": 323.0,
    "top": 81.0,
    "width": 96.0
   }
  ]
 },
 {
  "filename": "2.png",
  "boxes": [
   {
    "height": 32.0,
    "label": 2.0,
    "left": 77.0,
    "top": 29.0,
    "width": 23.0
   },
   {
    "height": 32.0,
    "label": 3.0,
    "left": 98.0,
    "top": 25.0,
    "width": 26.0
   }
  ]
```


Figure 1 Json file structure

Each image in the json file is represented by a dictionary having 2 keys:

1- Filename: which is the image's filename
2- Boxes: which is an array including the bounding boxes dimensions in each image, for each bounding box in an image it is characterized by the height, label, left, top, and width values.

Algorithm A

The pipeline of the algorithm will be described in detail with the methods used for each stage in the pipeline

1- First start by loading the needed libraries and packages

```
import os
import numpy as np
import pandas as pd
from collections import Counter
import cv2
import imutils
import json
import statistics
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import torch
from torch import nn
from torchvision import ops
from operator import itemgetter
```

Figure 2 Libraries and packages

2- Load training images and json file

```
images = []
labels = []
true_boxes = []

picsFolder_path = "train/"
with open('digitStruct.json') as f:
    data = json.load(f)

# import colored pictures
for i in range(len(data)):
    image = cv2.imread(picsFolder_path + data[i]['filename'])
    images.append(image)
    temp=[]
    for j in range(len(data[i]['boxes'])):
        temp.append(data[i]['boxes'][j]['label'])
    temp = np.array(temp)
    labels.append(temp)
    true_boxes.append(data[i]['boxes'])

print("we have",len(images),"images")
```

Figure 3 Loading training images and json file

i. Imports colored pictures from a folder ***"train/train/"***
ii. Store them in a list called images.
iii. Read the json file called ***"digitstruct.json"*** which contains information about the images and their associated bounding boxes.
iv. For each image, extract the labels of the digits in the image and store them in a list called ***"labels"*** (which will be used in phase 2 to indicate what each digit is)
v. Store the bounding box coordinates in a list called ***"true_boxes".***
vi. Print the total number of images that were loaded for verification only.

3- Resize all images

```
resizedImages =[]
for image in images:
    resizedImages.append(cv2.resize(image,(100,75)))
```

Figure 4 Images Resizing

Resizes all the images in the ***"images"*** list to a new size of 100 pixels width by 75 pixels height using OpenCV's **resize** function to standardize the size

of the images as they have different dimensions. The resized images are stored in a new list called ***"resizedimages".***

4- Make a copy of the resized images

```
imagesCopy = []
for resizedimage in resizedImages:
    imagesCopy.append(resizedimage)
```

Figure 5 Making a copy of resized images

Create a new list called ***"imagescopy"*** and copy all the images from the ***"resizedimages"*** list to this new list. Creating a backup of the images that will be needed to be restored later in the code

5- Convert original copy of the image from BGR to grayscale

```
greyImages = []
# Convert BGR to grayscale:
for resizedimage in resizedImages:
    greyImages.append(cv2.cvtColor(resizedimage, cv2.COLOR_BGR2GRAY))
```

Figure 6 Convert images from BGR to greyscale

Convert all the images in the ***"resizedimages"*** list from the bgr color space to grayscale using OpenCV's **cvtcolor** function. The grayscale images are stored in a new list called ***"greyimages".***

6- Carry out adaptive thresholding

```python
# Set the adaptive thresholding:
windowSize = 31
windowConstant = -1

binaryImages = []


# Apply the threshold:
for greyimage in greyImages:
    binaryImages.append(cv2.adaptiveThreshold(greyimage, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,
                                              windowSize, windowConstant))
```

Figure 7 Adaptive thresholding on greyscale images

Apply adaptive thresholding to images in ***"greyimages"*** using OpenCV's **adaptivethreshold** function. The adaptive thresholding is performed using a gaussian window with a size of 31x31 pixels (**windowsize**) and a constant offset of -1 (**windowconstant**). The resulting binary images are stored in a new list called ***"binaryimages"***. Adaptive thresholding allows the threshold to adapt to local lighting conditions since images' illumination varies across the image

7- Carry out connected components

```python
filteredImages =[]

# Perform Connected Components:
for binaryimage in binaryImages:
    componentsNumber, labeledImage, componentStats, componentCentroids = cv2.connectedComponentsWithStats(binaryimage,
                                                                                                           connectivity=4)
    # Set the minimum pixels for the area filter to filter connected components:
    minArea = 20

    # Get the indices/labels of the remaining components based on the area stat
    # (skip the background component at index 0)
    remainingComponentLabels = [i for i in range(1, componentsNumber) if componentStats[i][4] >= minArea]
    # Filter the labeled pixels based on the remaining labels,
    # assign pixel intensity to 255 (uint8) for the remaining pixels
    # one problem is that background connected components may still exist
    filteredImages.append(np.where(np.isin(labeledImage, remainingComponentLabels) == True, 255, 0).astype('uint8'))
```

Figure 8 Connected components on binary images

Perform connected component analysis on a set of binary images from the grayscale images in ***"greyimages".***

i. Call the **connectedcomponentswithstats** function from OpenCV to identify the connected components in each binary image. The function returns various information about each connected component, such as its size and centroid.

ii. Filter the connected components based on their area using a minimum area threshold (set to 20 pixels here). Connected components with an area below this threshold are discarded, since they are likely to correspond to noise or artifacts rather than digits.
iii. Create a new image by assigning a pixel intensity of 255 (white) to the remaining connected components, and 0 (black) to all other pixels. The resulting image is stored in a list called ***"filteredimages".***

note: some background connected components may still exist

8- Carry out morphological closing operation

```
# Set kernel (structuring element) size:
kernelSize = 3

# Set operation iterations:
opIterations = 1

# Get the structuring element:
maxKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (kernelSize, kernelSize))

closingImages = []

# Perform closing:
for filteredimage in filteredImages:
    closingImages.append(cv2.morphologyEx(filteredimage, cv2.MORPH_CLOSE, maxKernel, None, None, opIterations,
                                          cv2.BORDER_REFLECT101))
```

Figure 9 Morphological closing operation

Perform morphological closing on binary images in ***"filteredimages"*** using OpenCV's **morphologyex** function.

The closing operation is performed using a rectangular structuring element with a size of 3x3 pixels (**kernelsize**). The closing operation is applied to each binary image in ***"filteredimages"*** using the **cv2.morph_close** flag. The number of iterations of the closing operation is set to 1 (**opiterations**).

The resulting images are stored in a new list called ***"closingimages"***

9- Detect edges

```
edgeImages =[]

# perform smoothing
for closingimage in closingImages:
# perform canny edge detection:
    edgeImages.append(cv2.Canny(closingimage, 100, 200))
```

Figure 10 Canny Edge detection

Apply the canny edge detection algorithm to the binary images in ***"closingimages"*** using OpenCV's **canny** function.

The canny edge detection is performed on each binary image in ***"closingimages"*** using two threshold values, 100 and 200, specified as the second and third arguments to the canny function. These values determine the lower and upper thresholds for edge detection. Pixels with gradient magnitudes above the upper threshold are considered edges, while pixels with gradient magnitudes below the lower threshold are not considered edges. Pixels with gradient magnitudes between the two thresholds are only considered edges if they are connected to pixels above the upper threshold.

The resulting edge images are stored in a new list called ***"edgeimages".***

10- Draw bounding boxes on digits

```python
# index = 0
# Get each bounding box
# Find the big contours on the filtered image:
# for edgeimage in edgeImages:
def draw_boxes(edgeimage, copyimage):
    contours, hierarchy = cv2.findContours(edgeimage, cv2.RETR_CCOMP, cv2.CHAIN_APPROX_SIMPLE)
    contours_poly = [None] * len(contours)
    # The Bounding Rectangles will be stored here:
    boundRect = []
    # Alright, just look for the outer bounding boxes:
    for i, c in enumerate(contours):
        if hierarchy[0][i][3] == -1:
            contours_poly[i] = cv2.approxPolyDP(c, 3, True)
            boundRect.append(cv2.boundingRect(contours_poly[i]))
    for i in range(len(boundRect)):

        color = (0, 255, 0)
        # filter contours according to the area of the rectangle (parameter re5em)
        if ( int(boundRect[i][2])*int(boundRect[i][3])>100 and int(boundRect[i][2])*int(boundRect[i][3])<1000):
            cv2.rectangle(copyimage, (int(boundRect[i][0]), int(boundRect[i][1])),
                          (int(boundRect[i][0] + boundRect[i][2]), int(boundRect[i][1] + boundRect[i][3])), color, 2)
    # index= index + 1
    boxes= []
            # Iterate through each bounding box
    for i in range(len(boundRect)):
        # Extract the coordinates and dimensions of the bounding box
        x, y, w, h = boundRect[i]

        # Append the bounding box to the list in the format of a dictionary
        boxes.append({'left': x, 'top': y, 'width': w, 'height': h})

    # Return the list of bounding boxes
    return boxes
```

Figure 11 Draw Bounding Boxes

The function **draw_boxes()** takes in an edge-detected image from ***"edgeimages"*** and a copy of the original image from ***"imagescopy"*** and returns a list of bounding boxes of detected objects in the image. The function does the following steps:

i- Find contours in the edge-detected image and then approximates the contours with polygons.

ii- For each polygon, the function calculates a bounding rectangle and appends it to a list of bounding rectangles.

iii- Iterate through each bounding rectangle, filter out those with area smaller than 100 or larger than 1000, and draw a green rectangle around the remaining bounding rectangles on the copy of the original image.

iv- Extract the coordinates and dimensions of each bounding rectangle, append them to a list in the format of a dictionary, and return the list of bounding boxes.

Extra preprocessing:

We also added some preprocessing to the algorithm in order to help increase the accuracy

1- Remove the background using 2 different techniques ,
2- Edge sharpening
3- Remove noise by gaussian blur
4- Morphological open operation

Overall pipeline will not be changed but before converting the images from bgr to greyscale we will remove the background

1- Remove background from images

In order to help make the digits independent on their backgrounds to enhance the localization output of the algorithm.

Background remover 1:

```
backgroundremovedImages=[]

for sharpenedimage in sharpenedImages:
    #== Parameters
    BLUR = 11
    CANNY_THRESH_1 = 10
    CANNY_THRESH_2 = 100
    MASK_DILATE_ITER = 10
    MASK_ERODE_ITER = 10
    MASK_COLOR = (0.0,0.0,1.0) # In BGR format
    grayimage = cv2.cvtColor(resizedimage,cv2.COLOR_BGR2GRAY)
     #-- Edge detection
    edges = cv2.Canny(grayimage, CANNY_THRESH_1, CANNY_THRESH_2)
    edges = cv2.dilate(edges, None)
    edges = cv2.erode(edges, None)
     #-- Find contours in edges, sort by area
    contour_info = []
    contours, _ = cv2.findContours(edges, cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)
    if len(contours) > 0:
        contour_info = []
        for c in contours:
            contour_info.append((
                c,
                cv2.isContourConvex(c),
                cv2.contourArea(c),
            ))
        contour_info = sorted(contour_info, key=lambda c: c[2], reverse=True)
        max_contour = contour_info[0]
        #-- Create empty mask, draw filled polygon on it corresponding to largest contour ----
        # Mask is black, polygon is white
        mask = np.zeros(edges.shape)
        cv2.fillConvexPoly(mask, max_contour[0], (255))
        #-- Smooth mask, then blur it
        mask = cv2.dilate(mask, None, iterations=MASK_DILATE_ITER)
        mask = cv2.erode(mask, None, iterations=MASK_ERODE_ITER)
        mask = cv2.GaussianBlur(mask, (BLUR, BLUR), 0)
        mask_stack = np.dstack([mask]*3)    # Create 3-channel alpha mask
        #-- Blend masked img into MASK_COLOR background
        mask_stack  = mask_stack.astype('float32') / 255.0
        masking= resizedimage.astype('float32') / 255.0
        masked = (mask_stack * masking) + ((1-mask_stack) * MASK_COLOR)
        masked = (masked * 255).astype('uint8')
        backgroundremovedImages.append(masked)

    else:
        # Handle case when no contours are detected, for example:
        print("No contours detected in image")
        backgroundremovedImages.append(resizedimage)
```

Figure 12 Background remover 1

A number of parameters are set for the background removal process, such as the degree of edge detection and the amount of erosion and dilation to be applied to the mask.

For each image in ***“resizedimages”***, the following steps are performed:

i. The image is converted to grayscale using **cv2.cvtcolor().**
ii. Edge detection is performed on the grayscale image using the canny edge detector in **cv2.canny().**
iii. The resulting edge image is dilated and eroded to reduce noise and improve the contour detection in **cv2.dilate() and cv2.erode().**
iv. The contours are detected in the edge image using **cv2.findcontours().**
v. If any contours are detected in the image, the largest contour is found and its convex hull is used to create a mask that will be applied to the original image in order to isolate the foreground in **cv2.fillconvexpoly().**
vi. The mask is then smoothed and blurred using dilation, erosion, and gaussian blur **in cv2.dilate(), cv2.erode(),** and **cv2.gaussianblur().**
vii. A 3-channel alpha mask is created using **np.dstack(),** and the original image is normalized to a float32 format and divided by 255.
viii. The original image is then multiplied by the mask and the background color, and then the resulting image is scaled back up to a uint8 format in order to produce the final masked image in **cv2.multiply().**
ix. If no contours are detected in the image, the original resized image is simply appended to ***"backgroundremovedimages"*** list.
x. Finally, the list ***"backgroundremovedimages"*** is returned with all of the processed images.

Background remover 2 (MOG):

```
backgroundremovedImages=[]

for resizedimage in resizedImages:

    # convert image into grayscale
    gray = cv2.cvtColor(resizedimage, cv2.COLOR_BGR2GRAY)

    # Create a background subtractor object
    bgSubtractor = cv2.createBackgroundSubtractorMOG2()

    # Apply the background subtractor to the grayscale image
    fgMask = bgSubtractor.apply(gray)

    # Apply morphological transformations to remove noise and fill holes
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
    fgMask = cv2.morphologyEx(fgMask, cv2.MORPH_OPEN, kernel)
    fgMask = cv2.morphologyEx(fgMask, cv2.MORPH_CLOSE, kernel)

    # Invert the mask to obtain the foreground
    fgMask = cv2.bitwise_not(fgMask)

    # Apply the mask to the original image
    fgImg = cv2.bitwise_and(resizedimage, resizedimage, mask=fgMask)

    # Display the result
    backgroundremovedImages.append(fgImg)
```

Figure 13 Background remover 2

MOG2 (mixture of gaussians) to the grayscale version of the input image.

The code takes the list ***"resizedimages"*** and applies background subtraction to each image as follows:

i. Converts each input image to grayscale using **cv2.cvtcolor**.
ii. Creates a background subtractor object using **cv2.createbackgroundsubtractormog2()** and applies it to the grayscale image using the `apply` method. The result is a binary mask where the foreground objects are represented by white pixels and the background is represented by black pixels.
iii. Apply morphological transformations to the mask using **cv2.morphologyex** to remove noise and fill holes in the foreground objects.
iv. Invert the mask using **cv2.bitwise_not** to obtain the foreground objects as white pixels on a black background.
v. Apply the mask to the original image using `cv2.bitwise_and` to obtain an image where the foreground objects are isolated from the background. The resulting images are stored in a list called ***"backgroundremovedimages***", which is returned by the code.

2- Edge Sharpening

```
sharpenedImages = []

for greyimage in greyImages:

    # apply Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(greyimage, (3, 3), 0)

    # apply Laplacian filter to extract edges
    laplacian = cv2.Laplacian(blurred, cv2.CV_64F)
    laplacian = cv2.convertScaleAbs(laplacian)


    # apply edge sharpening using the grayscale original image and the Laplacian edges
    sharpened = cv2.convertScaleAbs(greyimage - laplacian)
    sharpenedImages.append(sharpened)

print(len(sharpenedImages))
```

Figure 14 Edge Sharpening

It takes the list of grayscale images in ***"greyimages*** and performs edge sharpening on each image using a laplacian filter. Here are the steps involved:

i. Loop over each image in ***"greyimages".***
ii. Apply **gaussianblur** function to the current image using a kernel size of (3, 3) to reduce noise .
iii. Apply a laplacian filter to the blurred image to extract edges. The **cv2.laplacian** function takes the image and the data type **(`cv2.cv_64f`)** as inputs and returns the filtered image.
iv. Convert the laplacian filtered image to an absolute scale (positive values only) using **cv2.convertscaleabs**.
v. Perform edge sharpening using the grayscale original image and the laplacian edges. This is done by subtracting the laplacian image from the grayscale image and then scaling the result to an absolute value. The result image is added to the ***"sharpenedimages"*** list.

3- Morphological opening operation

```
# Set kernel (structuring element) size:
kernelSize = 3

# Set operation iterations:
opIterations = 1

# Get the structuring element:
maxKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (kernelSize, kernelSize))

openingImages = []

# Perform closing:
for closingimage in closingImages:
    openingImages.append(cv2.morphologyEx(closingimage, cv2.MORPH_CLOSE, maxKernel, None, None, opIterations,
                                          cv2.BORDER_REFLECT101))
```

✓ 1.3s

Figure 15 Morphological Opening operation

performs morphological opening on a list ***"closingimages"***

i. Sets the size of the kernel (structuring element) used for the morphological operations using `**kernelsize**` and the number of iterations of the operations using `**opiterations**`. Then get the structuring element for the morphological operations using **cv2.getstructuringelement**, specifying a rectangular kernel of size **`kernelsize`.**
ii. Creates an empty list called ***"openingimages"*** to store the output images.
iii. Apply morphological opening to each input image in the list using **cv2.morphologyex,** with the structuring element, number of iterations, and border mode **cv2.border_reflect101**. The output images are stored in the ***"openingimages"*** list.

4- Gaussian blur before edge detection and after morphological closing operation

```
smoothedImages =[]

# perform smoothing
for closingimage in closingImages:
    smoothedImages.append(cv2.GaussianBlur(closingimage, (3, 3), 0))
```

Figure 16 Gaussian Blur

Smoothing steps done:

i. Apply gaussian blur to ***"closingimages"*** using the **cv2.gaussianblur** function, with a kernel size of (3, 3) and a standard deviation of 0 in the x and y directions.

ii. Append the smoothed image to the ***"smoothedimages"*** list and return it

Algorithm 2

```python
def finalModel(image):

    boxes = []

    #convert the image to greyscale
    image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    #increase the contrast
    cv2.convertScaleAbs(image, image)

    #apply gaussian blur to smooth the image
    image = cv2.GaussianBlur(image, (3, 3), 0)

    #apply adaptive threshholding
    image = cv2.adaptiveThreshold(image,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,11,1)

    #apply canny edge detection
    image = cv2.Canny(image, 150, 200, 255)

    #find contours in image
    contours, hierarchy = cv2.findContours(image, cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)

    #get the center of the image
    h, w = image.shape
    center = (int(w/2), int(h/2))

    #calculate the bounding rectangle of each contour and add it to a dictionary, only if it's near the center
    for i in range(len(contours)):
        x, y, w, h = cv2.boundingRect(contours[i])
        if abs((x + w/2) - center[0]) < 1.5*w and abs((y + h/2) - center[1]) < 1.5*h:
            boxes.append({'left': x, 'top': y, 'width': w, 'height': h})


    return boxes
```

Figure 17 Algorithm 2

Algorithm steps with output of each step:

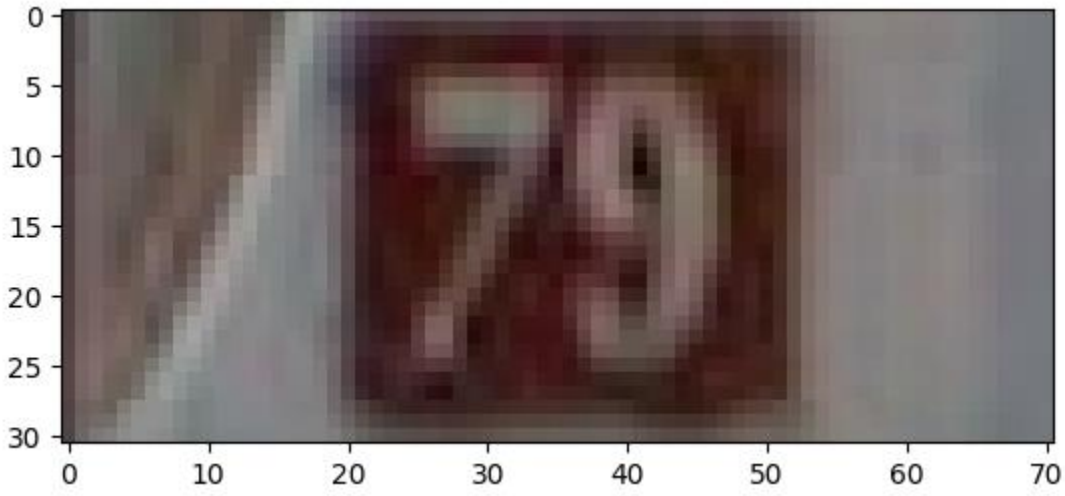

Figure 18 Original Image

i. Converts the input image to grayscale using **cv2.cvtcolor**

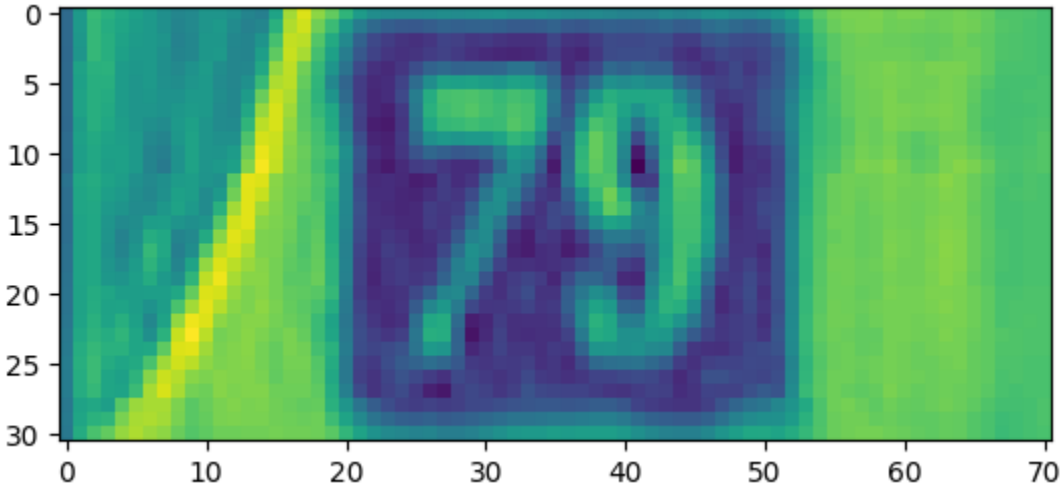

Figure 19 Greyscale image

ii. Increases the image contrast using **cv2.convertscaleabs**.

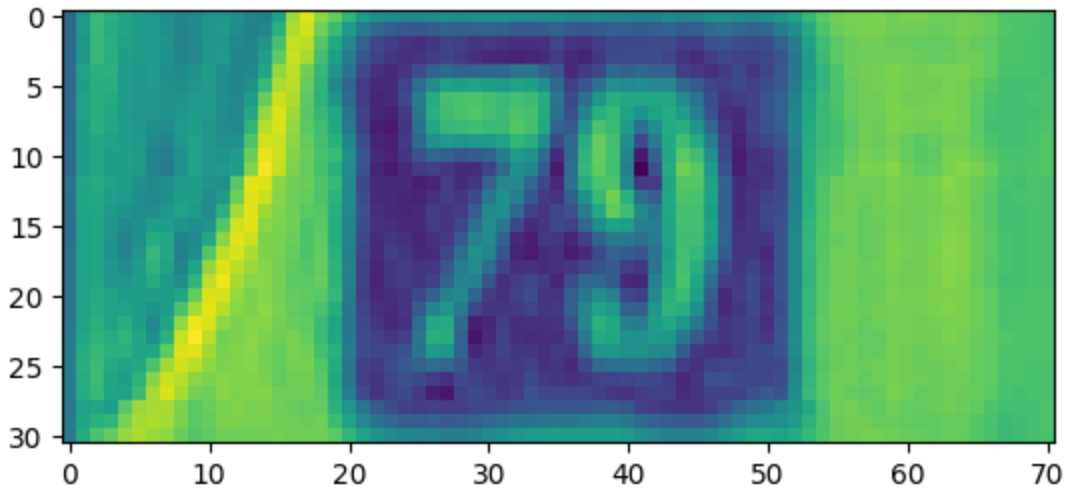

Figure 20 High Contrast Image

iii. Applies a gaussian blur to smooth the image using **cv2.gaussianblur**.

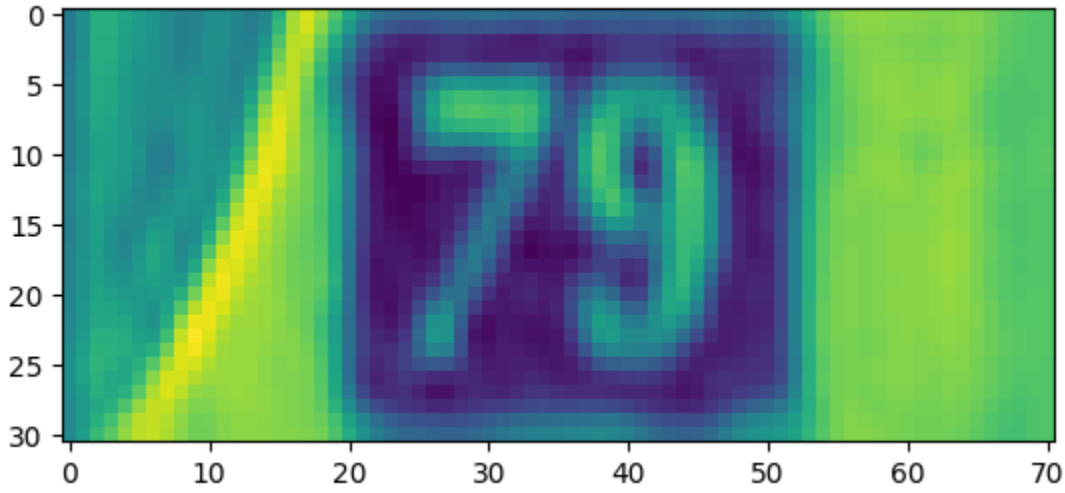

Figure 21 Gaussian Smoothed Image

iv. Applies adaptive thresholding to the image using **cv2.adaptivethreshold**. This technique adjusts the threshold level for each pixel in the image based on the local neighborhood, which can help to improve the contrast between objects and their background.

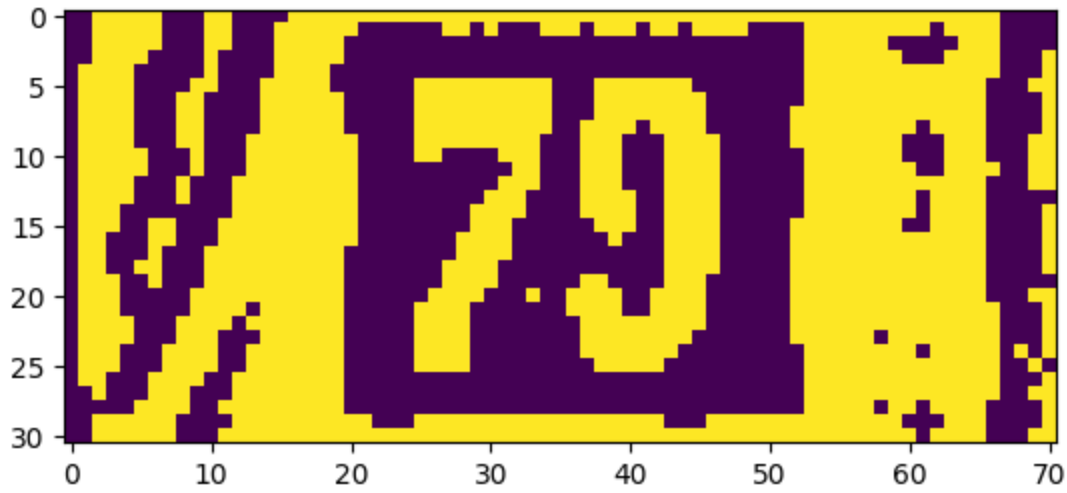

Figure 22 Adaptive Thresholded Image

v. Applies canny edge detection using **cv2.canny** to extract the edges of the objects in the image

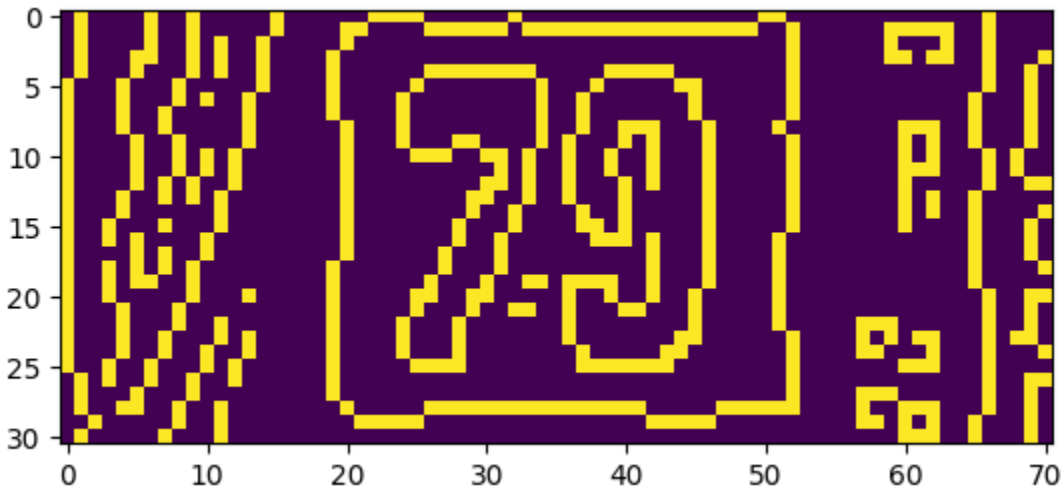

Figure 23 Edge detected Image

vi. Use **cv2.findcontours** to identify the contours in the image, which represent the boundaries of the objects.

vii. Calculates the center of the image and iterates over the contours to extract the bounding boxes of objects that are located close to the center. The **cv2.boundingrect** function is used to calculate the coordinates of the bounding box.

viii. Return a list of dictionaries, where each dictionary corresponds to a detected object and contains the coordinates of the bounding box as keys ('left', 'top', 'width', 'height').

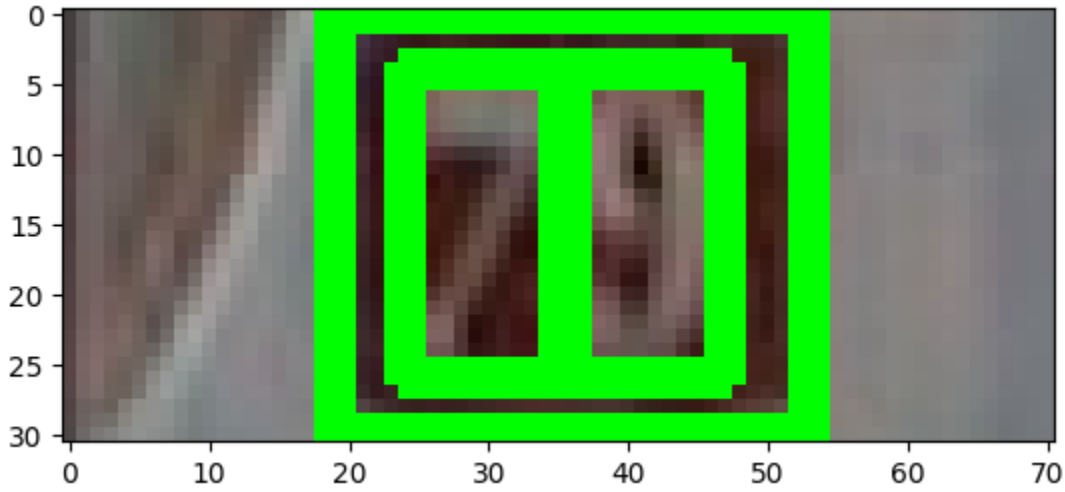

Figure 24 Final Output Image with bounding boxes

Accuracy measure

Intersection over union

```python
def box_iou(boxA, boxB):
    # Calculate the intersection coordinates of the two bounding boxes
    x1 = max(boxA[0], boxB[0])
    y1 = max(boxA[1], boxB[1])
    x2 = min(boxA[2], boxB[2])
    y2 = min(boxA[3], boxB[3])

    # Compute the area of intersection rectangle
    intersection_area = max(0, x2 - x1 + 1) * max(0, y2 - y1 + 1)

    # Compute the area of both bounding boxes
    boxA_area = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
    boxB_area = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)

    # Compute the union area
    union_area = boxA_area + boxB_area - intersection_area

    # Compute the IOU
    iou = intersection_area / union_area

    return iou
```

Figure 25 IOU

This function **box_iou** calculates the intersection over union (**IOU**) of two bounding boxes represented by four coordinates: `(x1, y1, x2, y2)`, where

`(x1, y1)` is the top-left coordinate and `(x2, y2)` is the bottom-right coordinate.

Steps of the function are:

i- Calculate the intersection coordinates of the two bounding boxes by finding the maximum `x1` and `y1` values and the minimum `x2` and `y2` values between the two boxes. Then calculate the area of the intersection rectangle using the intersection coordinates.

ii- Calculate the area of both bounding boxes by finding the difference between the maximum and minimum `x` and `y` coordinates of each box and adding 1 to each side to account for the edges of the box.

iii- Using these areas, compute the union area by adding the areas of both bounding boxes and subtracting the intersection area.

iv- calculate the IOU by dividing the intersection area by the union area.

The function returns the IOU value as a floating-point number between 0 and 1. If the IOU is 0, it means the two bounding boxes do not overlap at all, and if it is 1, it means the two bounding boxes are exactly the same.

IOUPicTest

```
def iouPicTest(truth, predicted, threshold1=0.5):
    ious = []
    for i in range(len(truth)):
        for j in range(len(predicted)):
            truth_box = [truth[i]['left'], truth[i]['top'], truth[i]['left'] + truth[i]['width'], truth[i]['top']+truth[i]['height']]
            predicted_box = [predicted[j]['left'], predicted[j]['top'], predicted[j]['left']+predicted[j]['width'], predicted[j]['top']+predicted[j]['height']]
            iou = box_iou(truth_box, predicted_box)
            if iou >= threshold1:
                ious.append(iou)
    acc = np.average(ious) if ious else 0
    return acc
```

Figure 26 IOUPicTest

This function **IOUpictest** takes in two lists of dictionaries representing ground truth bounding boxes and predicted bounding boxes for digit detected in an image. The function uses the **box_IOU** function defined above to calculate the IOU of each pair of truth and predicted bounding boxes.

Steps of function:

i- Loops through each pair of bounding boxes in `truth` and `predicted` and converts each bounding box from a dictionary with `left`, `top`,

`width`, and `height` keys to a list of four coordinates in the format expected by the **box_IOU** function.

ii- Calculate the IOU of the pair of bounding boxes using **box_IOU**.if the IOU value is greater than or equal to a given threshold value `threshold1` (default 0.5), the IOU value is added to a list of IOUs `IOUs`. If there are no IOUs above the threshold, the function returns an accuracy of 0.

iii- Calculate the average of the `IOUs` list using NumPy's **np.average** function and returns this value as the accuracy of the predicted bounding boxes. If the `IOUs` list is empty, meaning no predicted bounding boxes had an IOU above the threshold with any of the ground truth bounding boxes, it returns an accuracy of 0.

Algorithms Accuracy and output images

Algorithm 1

Accuracy :

Different accuracies computed for algorithm 1 by applying different preprocessing steps

```
acc = []
for i in range(0,len(images)):
    #plt.imshow(images[i])
    # plt.show()
    boxes = draw_boxes(edgeImages[i],imagesCopy[i])
    # tp,precision, recall, f1_score = overlapTest(edgeImages[i],imagesCopy[i])
    acc.append(iouPicTest(true_boxes[i],boxes))
    # print (acc)
    #plt.imshow('image',imagesCopy[i])
    # cv2.waitKey(0)
print(np.average(acc)*100)
# print(precision)
# print(recall)
# print(f1_score)
# print(tp/len(true_boxes))

✓ 14.9s
3.8058156365122526
```

Figure 27 Algorithm 1 accuracy 1

```
acc = []
for i in range (0,len(images)):
    #plt.imshow(images[i])
    #plt.show()
    boxes = draw_boxes(edgeImages[i],imagesCopy[i])
    # tp,precision, recall, f1_score = overlapTest(edgeImages[i],imagesCopy[i])
    acc.append(iouPicTest(true_boxes[i],boxes))
    # print (acc)
    #plt.imshow(imagesCopy[i])
    #plt.show()
    # cv2.waitKey(0)
print(np.average(acc)*100)
# print(precision)
# print(recall)
# print(f1_score)
# print(tp/len(true_boxes))
```

✓ 1m 53.1s

```
3.9038389325732568
```

Figure 28 Algorithm 1 accuracy 2

```
acc = []
for i in range(0,len(images)):
    # plt.imshow(images[i])
    # plt.show()
    boxes = draw_boxes(edgeImages[i],imagesCopy[i])
    # tp,precision, recall, f1_score = overlapTest(edgeImages[i],imagesCopy[i])
    acc.append(iouPicTest(true_boxes[i],boxes))
    # print (acc)
    # cv2.imshow('image',imagesCopy[i])
    # cv2.waitKey(0)
print(np.average(acc)*100)
# print(precision)
# print(recall)
# print(f1_score)
# print(tp/len(true_boxes))
```

✓ 18.0s

```
7.6504325340347705
```

Figure 29 Algorithm 1 accuracy 3

The maximum accuracy that can be reached by this algorithm is 7.6%

Sample of output images:

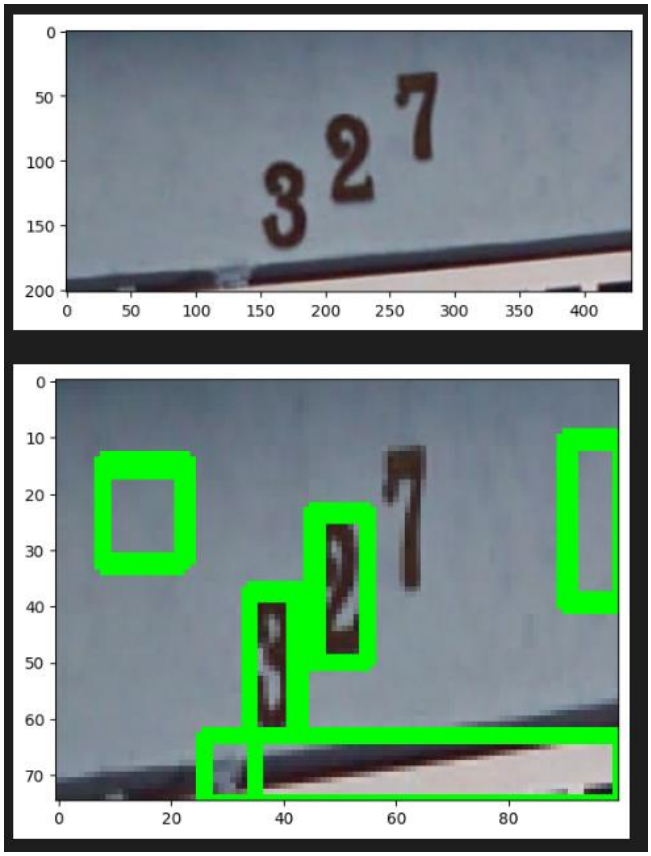

Figure 31 Algorithm 1 output 1

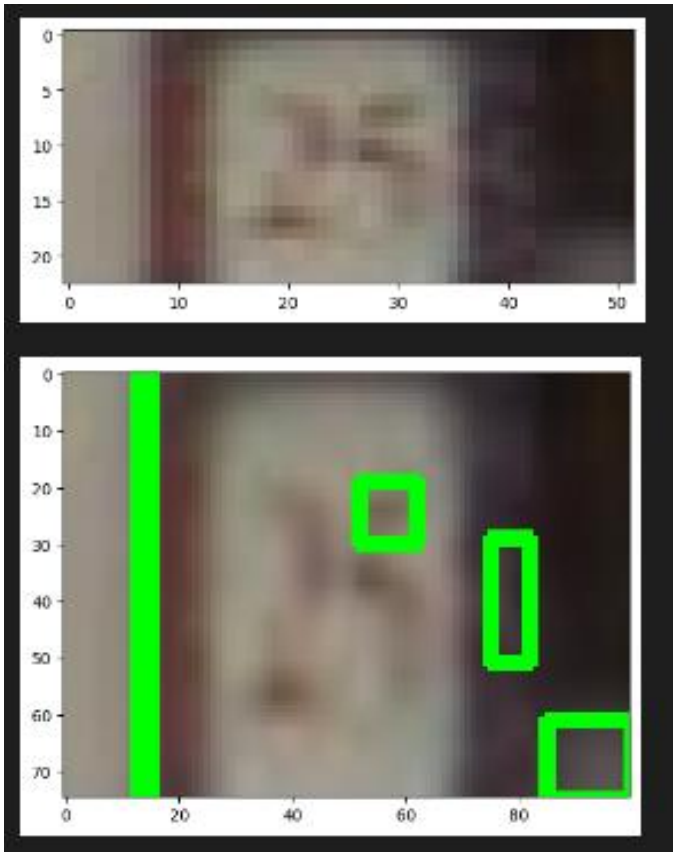

Figure 30 Algorithm 1 output 2

Figure 33 Algorithm 1 output 3

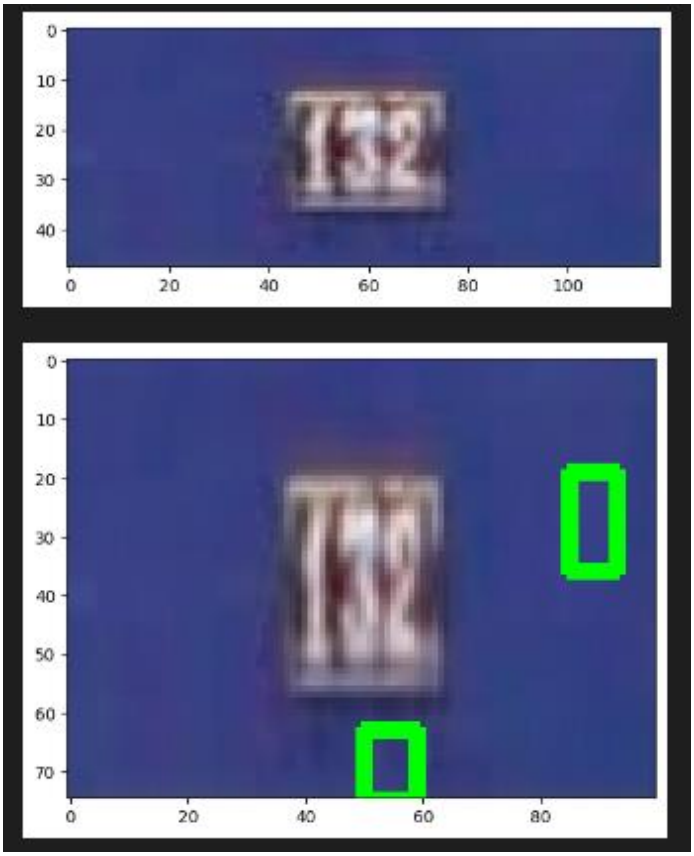

Figure 32 Algorithm 1 output 4

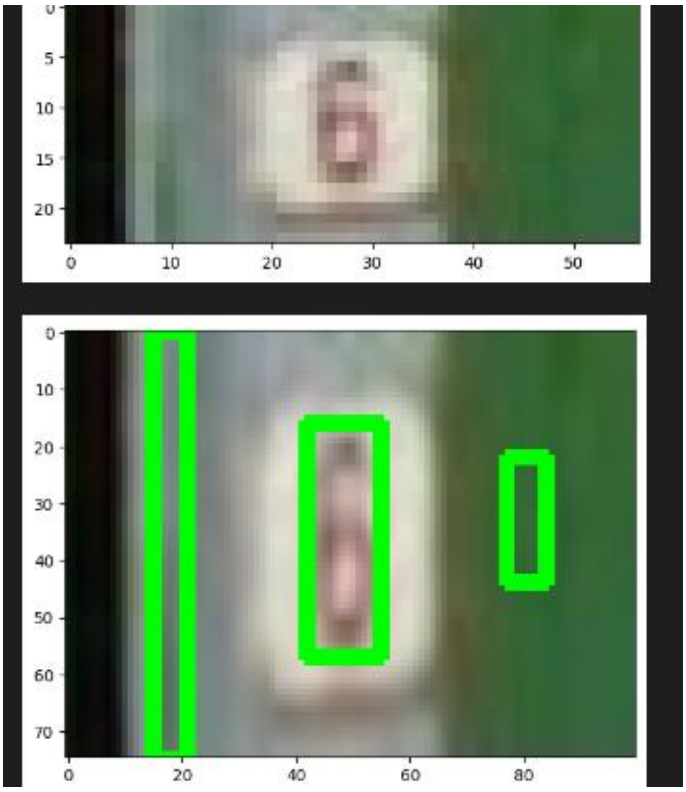

Figure 35 Algorithm 1 output 5

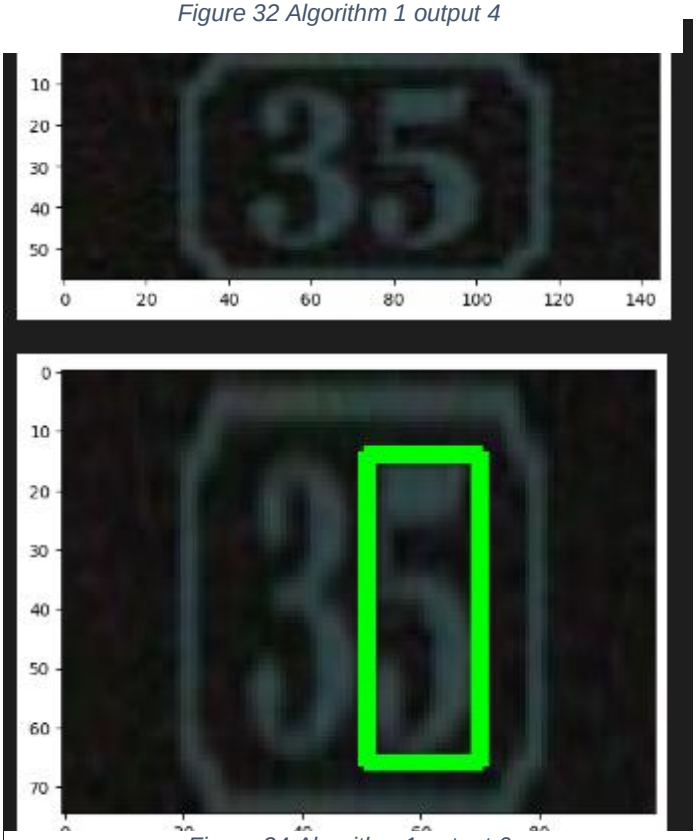

Figure 34 Algorithm 1 output 6

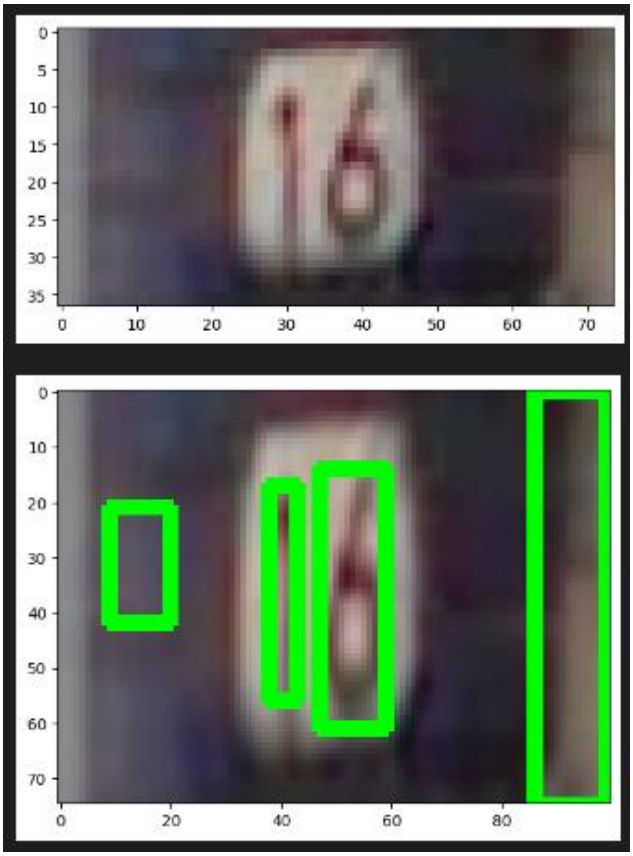

Figure 37 Algorithm 1 output 7

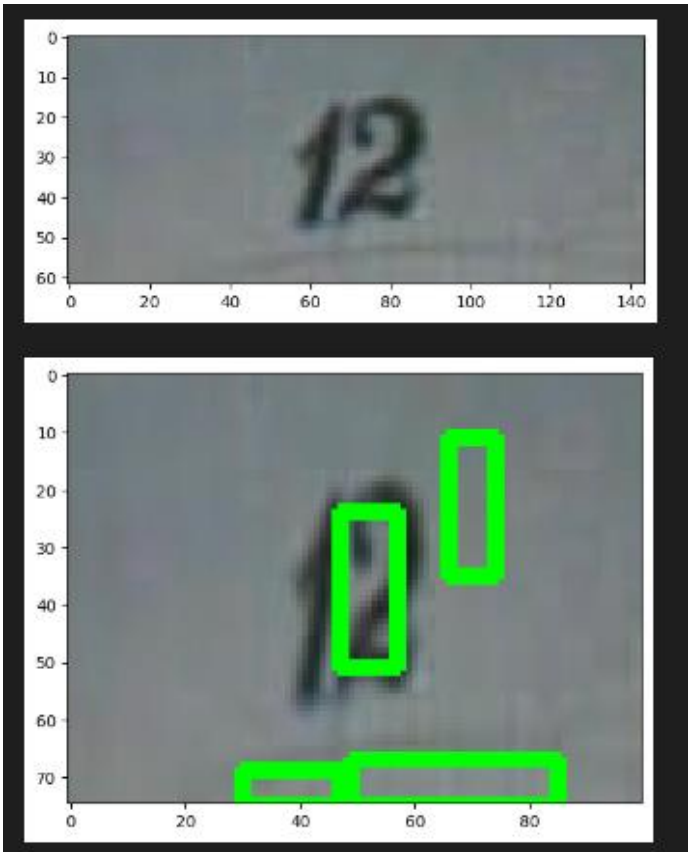

Figure 36 Algorithm 1 output 8

Algorithm 2

Accuracy :

The accuracy of algorithm 2 is 55%

```
acc = []

for i in range(0,len(images)):
    boxes = finalModel(images[i])
    acc.append(iouPicTest(true_boxes[i],boxes))
print(np.average(acc)*100)

for i in range(0,30):
    image = images[i].copy()
    predicted_boxes = finalModel(image)
    for i in predicted_boxes:
        cv2.rectangle(image, (i['left'], i['top']), (i['left'] +i['width'], i['top']+i['height']), (0, 255, 0), 2)
    #cv2.imshow('image',image)
    #cv2.waitKey(0)
```

✓ 2m 18.1s

```
55.046910655567004
```

Figure 38 Algorithm 2 accuracy

Sample of output images

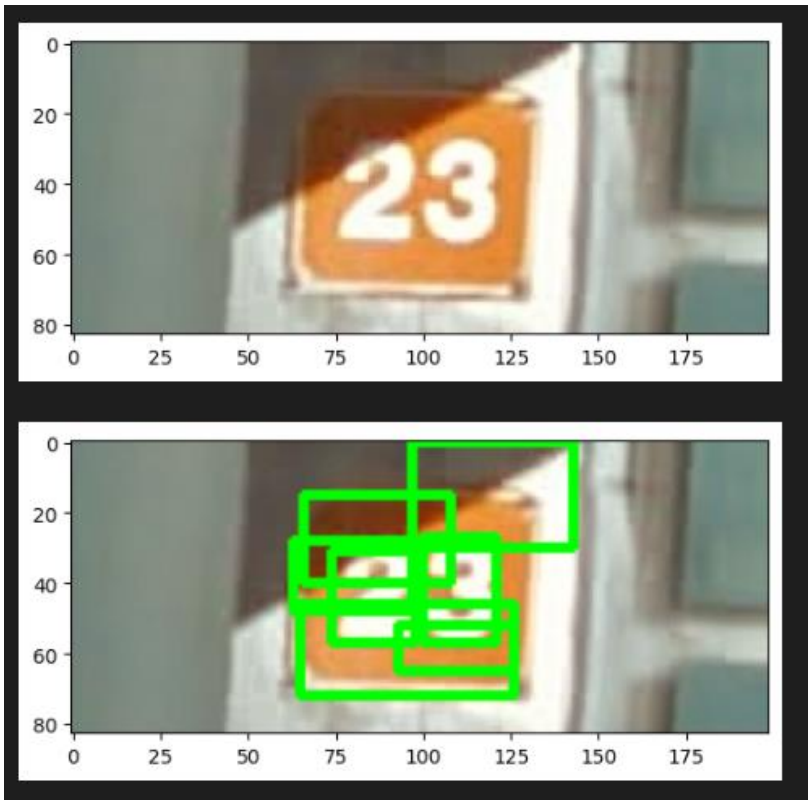

Figure 42 Algorithm 2 output 1

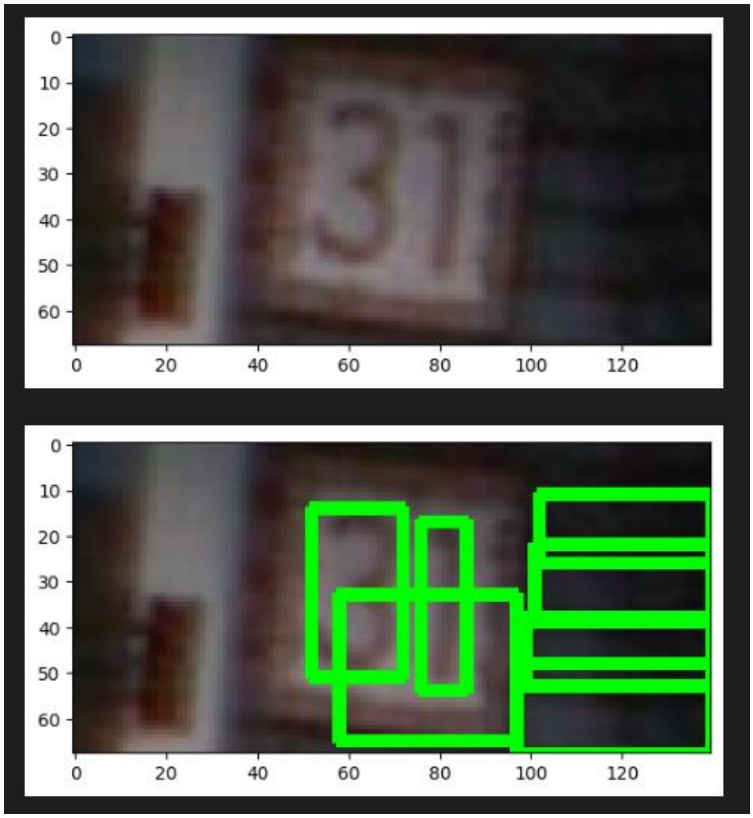

Figure 41 Algorithm 2 output 2

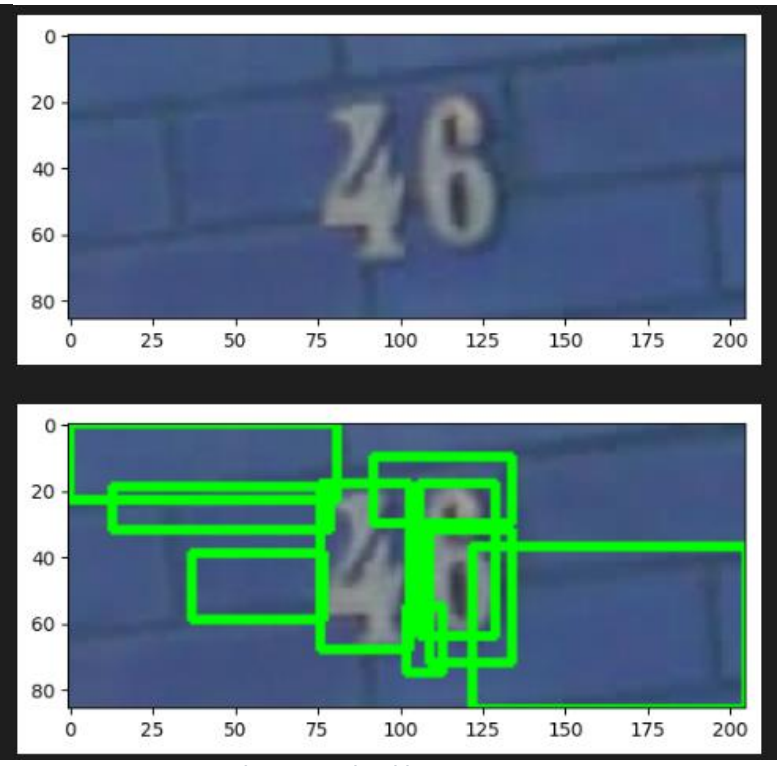

Figure 40 Algorithm 2 output 3

Figure 39 Algorithm 2 output 4

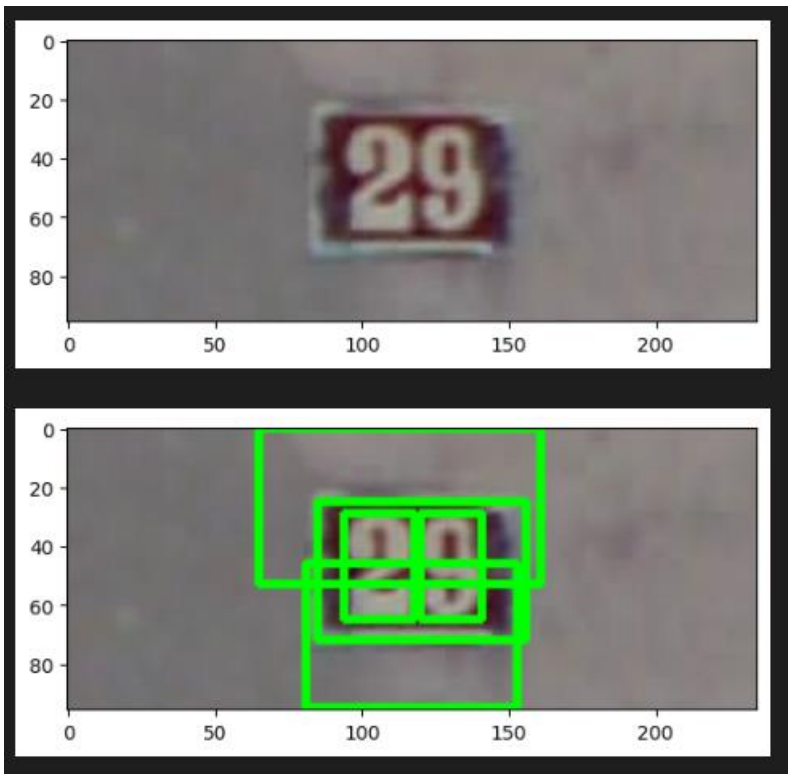

Figure 44 Algorithm 2 output 5

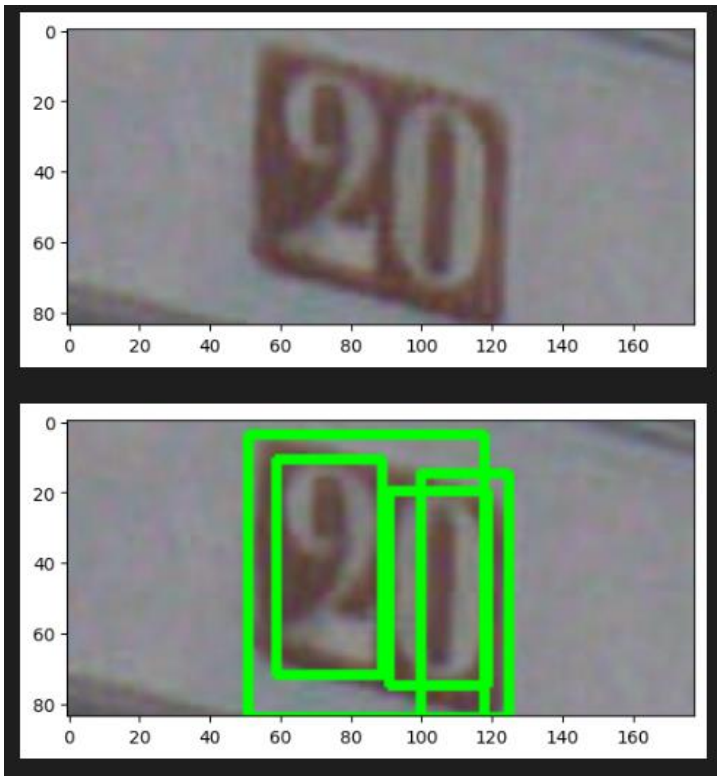

Figure 43 Algorithm 2 output 6

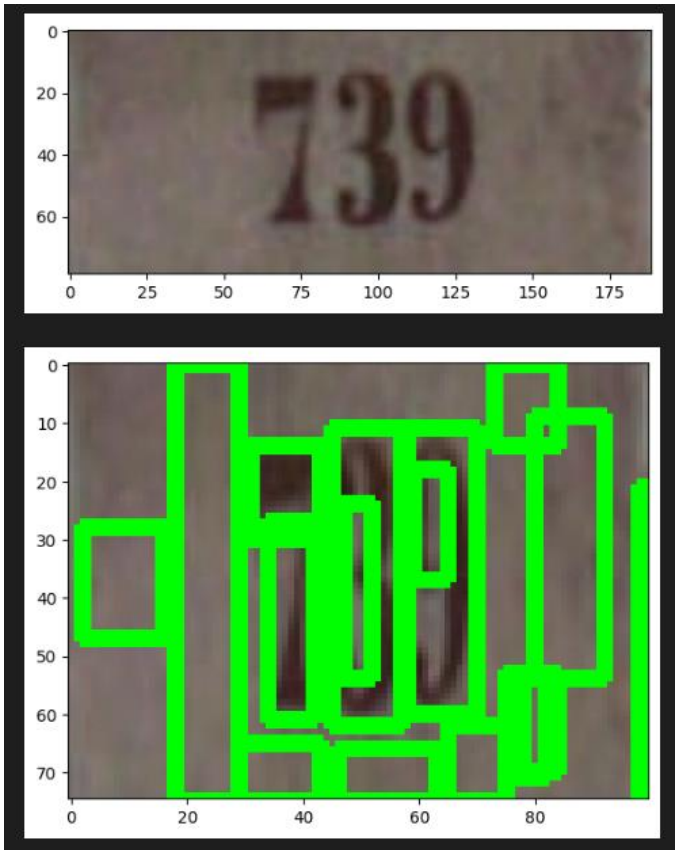

Figure 45 Algorithm 2 output 7

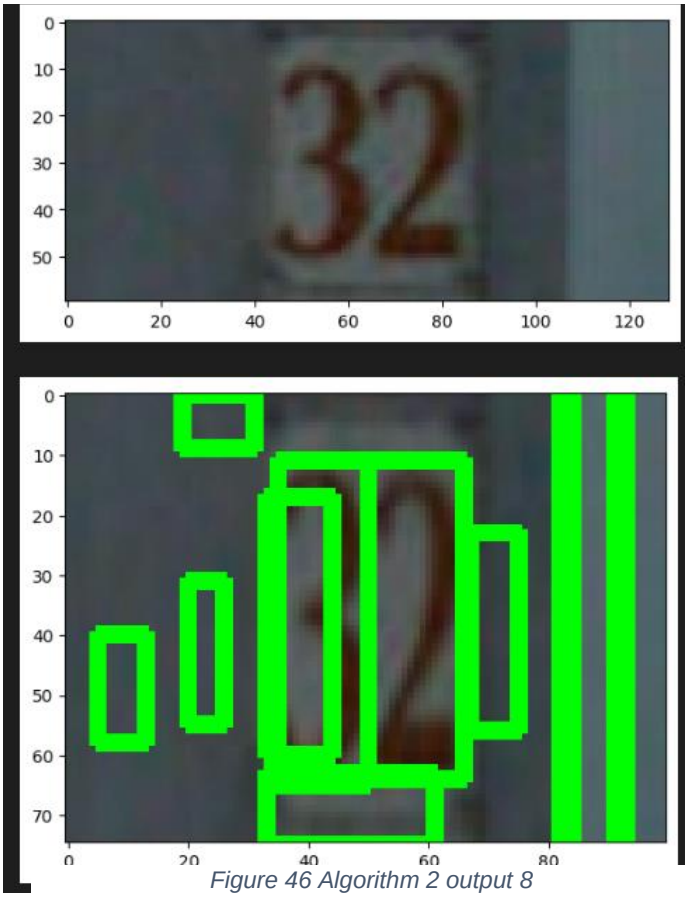

Figure 46 Algorithm 2 output 8

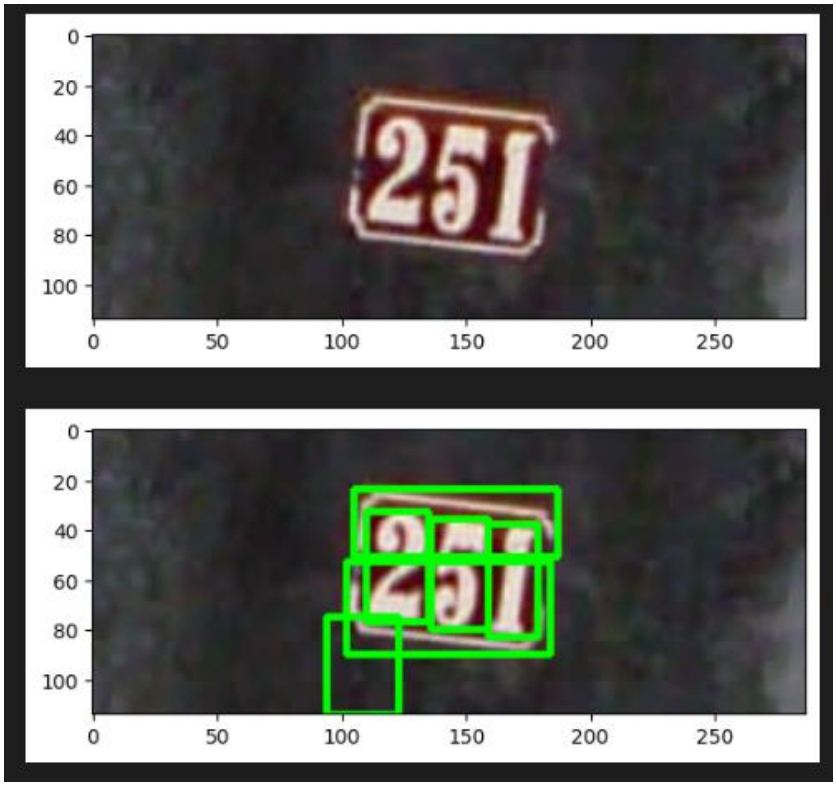

Figure 47 Algorithm 2 output 9

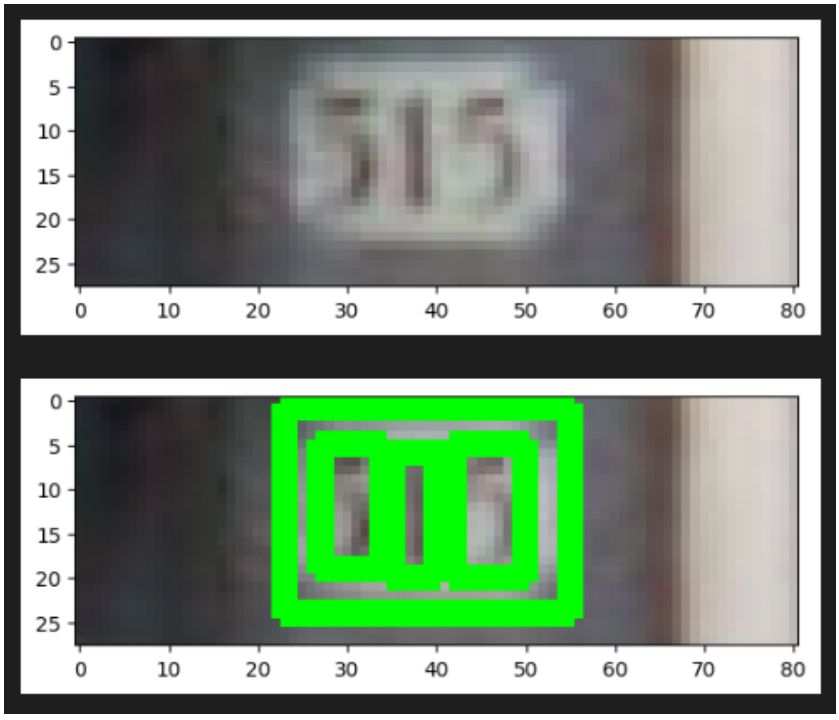

Figure 48 Algorithm 2 output 10

This algorithm takes longer to execute but however it has higher accuracy than the algorithm a mentioned earlier which makes it much better to use.

This algorithm detects more contours than the above algorithm which increases the chances to detect the digits correctly in the image from the extracted contours.

Conclusion

Using traditional computer vision to detect the digits in this dataset is a huge challenge as the images are variant and makes it difficult to choose the perfect hyperparameters for each function or method used that will optimize the results for all of the output images.

Our 2 algorithms, were tuned by try and error and by trying different preprocessing methods to get the highest accuracies as possible. However, the accuracy measure differs and therefore there it is difficult to decide whether or not the algorithm is optimal.

Github repository

https://github.com/aGayar30/computer-vision-project