Logo

Description automatically generated

CSE483 Computer Vision

**Project Documentation**

**Phase 1**

**Submitted to:**

Prof. Dr. Mahmoud Ibrahim Khalil

Eng. Mahmoud Soheil

**Submitted by:**

Youssef George Fouad 19P9824

Kerollos Wageeh Youssef 19P3468

Nada Amr Attia 19P1621

CESS Senior-1

Table of Contents

[1. Problem Description 3](#_Toc134117652)

[2. GETTING STARTED 3](#_Toc134117653)

[2.1. Importing Needed Libraries 3](#_Toc134117654)

[2.2. Importing Our Dataset 3](#_Toc134117655)

[3. PIPELINE WALKTHROUGH 4](#_Toc134117656)

[3.1. Bilateral Filter 4](#_Toc134117657)

[3.1.1. Choosing filter hyperparameters 5](#_Toc134117658)

[3.2. Convert RGB to Grayscale 5](#_Toc134117659)

[3.3. Enhance Image Contrast 5](#_Toc134117660)

[3.4. Adaptive Gaussian Thresholding 6](#_Toc134117661)

[3.5. (optional) Morphological Opening 6](#_Toc134117662)

[3.6. (optional) Morphological Closing 7](#_Toc134117663)

[3.7. (optional) Canny Edge Detection 7](#_Toc134117664)

[3.8. Find Contours 8](#_Toc134117665)

[3.9. Draw Boundary Boxes 8](#_Toc134117666)

[4. FULL MODEL 9](#_Toc134117667)

[5. ACCURACY METRICS 10](#_Toc134117668)

[5.1. Average Boxes Intersection Over Union (IOU) 10](#_Toc134117669)

[5.2. Boxes Recall 11](#_Toc134117670)

[6. TESTING SCENARIOS 11](#_Toc134117671)

[6.1. Single Image 11](#_Toc134117672)

[6.2. Full images folder 12](#_Toc134117673)

[6.3. Image Resolution Grouping 13](#_Toc134117674)

[6.4. Numbers-Area-to-Image-Size Ratio Grouping 16](#_Toc134117675)

[6.5. Image Brightness Grouping 19](#_Toc134117676)

[7. Source Code: 23](#_Toc134117677)

# Problem Description

SVHN is a real-world image dataset for developing object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST (e.g., the images are of small, cropped digits), but incorporates an order of magnitude more labelled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images.

The output of this phase is to localize the digits in the test images.

# GETTING STARTED

## Importing Needed Libraries

Before building our computer vision pipeline, we start by importing needed python libraries that help us process the images, test our algorithm accuracy, and visualize the results.

import numpy as np

import cv2

from scipy.io import loadmat

import json

import statistics

import torch

from torchvision import ops

from operator import itemgetter

import random

import tensorflow as tf

import matplotlib.pyplot as plt

## Importing Our Dataset

The dataset consists of images, labels, and boxes that define the ground truth localization for the numbers in the images. The RGB colored images are given in PNG format which are easy to read into python in shape. The other 2 are given in MAT file format that we pre-converted into JSON format to be easily read as dictionaries in python, where 1 dictionary represents 1 number in an image. A dictionary has 5 keys: label, left, top, width, and height. **,** , where these 2 points are 2 diagonal vertices of a box localizing a single digit in an image.

We defined the following function, “” that takes one argument to specify which data folder to import, train, test, or extra. It returns 3 arrays of the images, labels, boxes.

def getPics(chosen\_set):

images = []

labels = []

boxes = []

picsFolder\_path = "SVHN/" + chosen\_set + "/"

with open(picsFolder\_path + '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)

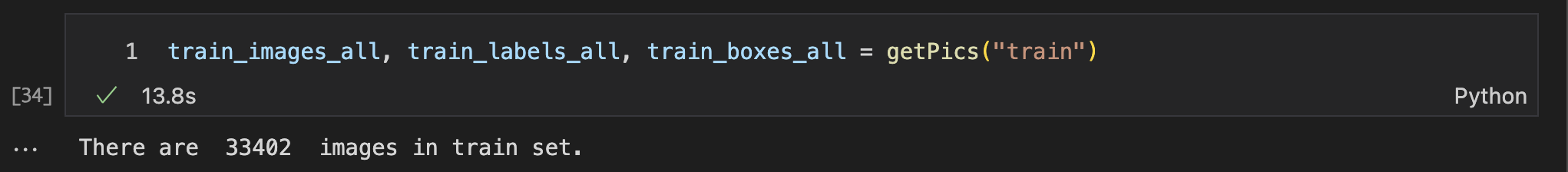
labels.append(temp)

boxes.append(data[i]['boxes'])

print("There are ", len(data), " images in " + chosen\_set + " set.")

return images, labels, boxes

For example:



# PIPELINE WALKTHROUGH

## Bilateral Filter

Our first step is applying bilateral filter on the RGB picture to blur out all the picture except the edges.

image = cv2.bilateralFilter(trialImage,11,7,7)

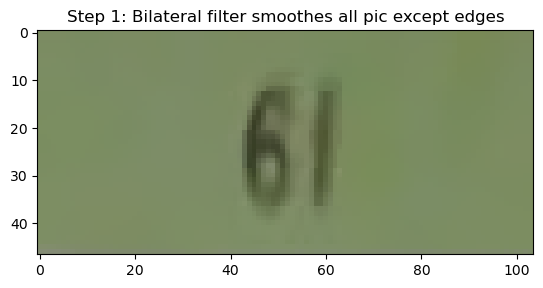
Example in figure 1 shows how bilateral filter reduces noise by blurring everything except for the edges.

Figure : Bilateral filter effect

### Choosing filter hyperparameters

In order to know the best possible hyperparameter, sigma value, of the bilateral filter, we run our full algorithm on each set of pictures, on a range of values, to check the sigma value resulting in the best localization accuracy. The results were as follow.

For the images in the training folder (33,402), the best accuracy was 46.5% at sigma = 12.

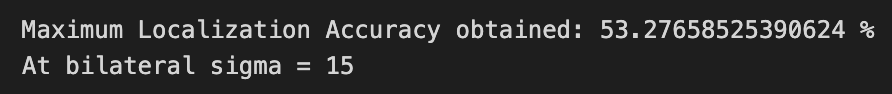
*Graph + res*

For the images in the test folder (13,068), the best accuracy was 43.1% at sigma = 17.

*Graph + res*

For the images in the test folder (first 50,000), the best accuracy was 53.3% at sigma = 15.

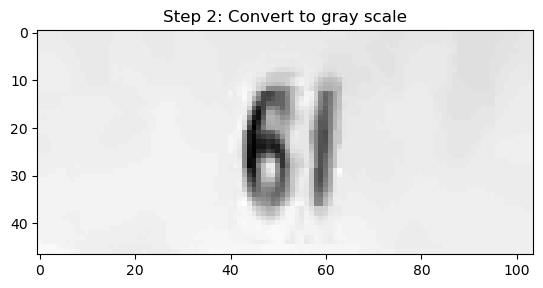
*Graph*



## Convert RGB to Grayscale

Reducing color channels from 3 (Red, Green, Blue) to a single gray channel (0-255), where 0 is black and 255 is white, is our second step.

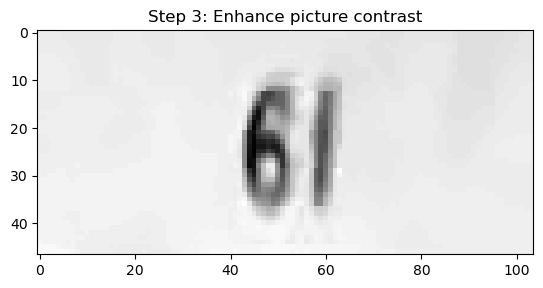
image = cv2.cvtColor(image.copy(),cv2.COLOR\_RGB2GRAY)



## Enhance Image Contrast

Enhancing image contrast equalizes the gray values histogram and helps in getting better results.

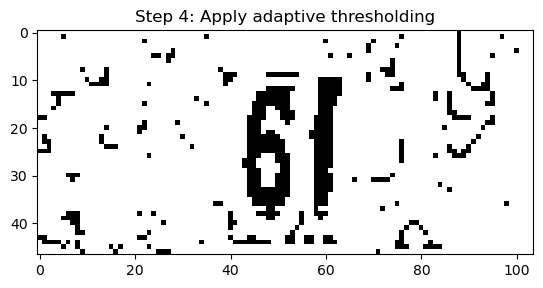
cv2.convertScaleAbs(image, image)



## Adaptive Gaussian Thresholding

Applying adaptive thresholding which divides the image into a grid of equal blocks and specifies a different threshold for each block. The output of this step is a binary image (0 or 1), where 0 represents a black pixel and 1 represents a white one.

bnr = cv2.adaptiveThreshold(image,255,cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY,11,1)



## (optional) Morphological Opening

As an optional step, we can use a 3x3 all-ones kernel to apply morphological opening to the binary image from the last step to reduce noise. This step is optional as it only works on a portion of the images set not all of them according to the image composition and quality.

kernel = np.ones((3, 3), np.uint8)

bnr = cv2.morphologyEx(bnr, cv2.MORPH\_OPEN, kernel, iterations=1)

Chart

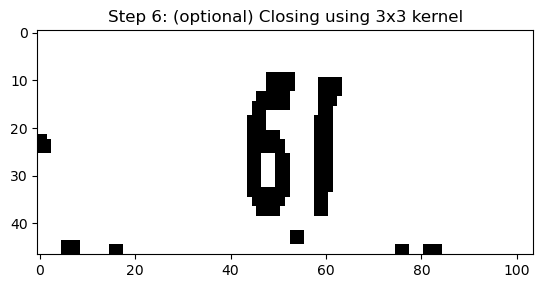
Description automatically generated

## (optional) Morphological Closing

Another optional step, we can use a 3x3 all-ones kernel to apply morphological closing to the binary image from the last step to close any gaps caused by the opening. This step is optional as it only works on a portion of the images set not all of them according to the image composition and quality.

kernel = np.ones((3, 3), np.uint8)

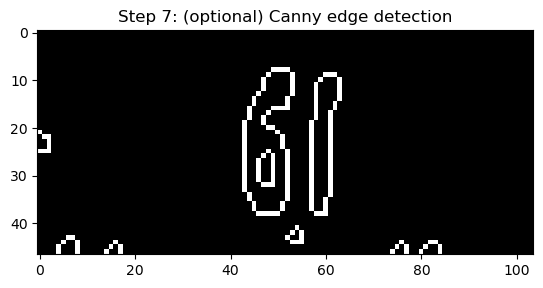
bnr = cv2.morphologyEx(bnr, cv2.MORPH\_CLOSE, kernel, iterations=1)



## (optional) Canny Edge Detection

Canny edge detection is considered an optional step here as a binary image would be enough to later find contours. We will only apply it here to see how it acts.

bnr = cv2.Canny(bnr, cannyTH1, cannyTH2, 255)



## Find Contours

Second to last step is finding the contours based on the binary image from our last step.

contours, hierarchy = cv2.findContours(bnr, cv2.RETR\_LIST, cv2.CHAIN\_APPROX\_NONE)

temp = trialImage.copy()

cv2.drawContours(temp,contours,-1,color=(0,255,0),thickness=1)

A picture containing graphical user interface

Description automatically generated

## Draw Boundary Boxes

Last step in our localization algorithm, we draw bounding boxes on the detected contours.

for i in range(len(contours)):

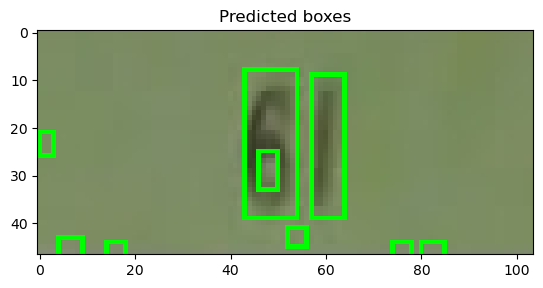
x,y,w,h = cv2.boundingRect(contours[i])

cv2.rectangle(temp,

(x,y),

(x+w,y+h),

(0,255,0), 1)



# FULL MODEL

Combining all the previously mentioned steps in one function that we called . It is fully customizable using some binary constants (0 or 1) to enable or disable some steps or even configure the hyperparameters.

The function takes a single image and returns the predicted boxes which can be later be assessed for accuracy or for visualization.

# CONSTANTS

useCONTRAST = 1

useCLAHE = 0

useOPEN = 0

useCLOSE = 0

useCANNY = 0

cannyTH1 = 150

cannyTH2 = 200

bilateralSigma = 12

# Fn performs CV techniques on a single picture

def rectanglesModel(img, bilateral=7):

image = img.copy()

boxes = []

kernel = np.ones((3, 3), np.uint8)

image = cv2.bilateralFilter(image,11,bilateral,bilateral)

image = cv2.cvtColor(image, cv2.COLOR\_RGB2GRAY)

# image = cv2.GaussianBlur(image, (3, 3),0)

if(useCONTRAST):

cv2.convertScaleAbs(image, image)

if(useCLAHE):

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))

image = clahe.apply(image)

bnr = cv2.adaptiveThreshold(image,255,cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY,11,1)

if(useOPEN):

bnr = cv2.morphologyEx(bnr, cv2.MORPH\_OPEN,

kernel, iterations=1)

if(useCLOSE):

bnr = cv2.morphologyEx(bnr, cv2.MORPH\_CLOSE,

kernel, iterations=1)

if(useCANNY):

bnr = cv2.Canny(image, cannyTH1, cannyTH2, 255)

contours, hierarchy = cv2.findContours(bnr, cv2.RETR\_LIST, cv2.CHAIN\_APPROX\_NONE)

for i in range(len(contours)):

boxes.append({})

x,y,w,h = cv2.boundingRect(contours[i])

boxes[i]['left'], boxes[i]['top'], boxes[i]['width'], boxes[i]['height'] = x,y,w,h

return boxes

# ACCURACY METRICS

After applying our algorithm on an image or a set of images, we need to test how accurate our localization technique was; therefore, we use the following 2 metrics to evaluate the results compared to the ground truth provided with the dataset.

## Average Boxes Intersection Over Union (IOU)

IOU is dividing the intersection area between two boxes over their union area. But that’s not as simple and direct as it sounds.

x = ops.box\_iou(truth\_box, predicted\_box)

1. Firstly, we performIOU **between each of the output boxes and the ground truth boxes** to determine which of our boxes are actually surrounding numbers. Predicted boxes that result in less than a certain , typically < are eliminated.

This results in finding all the boxes that surround numbers in the image.

1. A single number in an image can have multiple boxes surrounding parts or all of it. We perform non-maximum suppression to find which box best covers a number.
2. Weighted average of IOU values is calculated, to give a zero value to numbers that were not localized.
3. The built function returns the calculated accuracy in addition to the selected boxes.

*Full code as follows:*

# IOU average accuracy test per picture

def iouPicTest(truth, predicted, threshold1=0.5, threshold2=0.5):

filtered = []

boxesTensors = []

# Check IOU of predicted against all true boxes

for i in range(len(truth)):

for j in range(len(predicted)):

truth\_box = torch.tensor(

[[truth[i]['left'], truth[i]['top'], truth[i]['left'] + truth[i]['width'],

truth[i]['top']+truth[i]['height']]], dtype=torch.float)

predicted\_box = torch.tensor(

[[predicted[j]['left'], predicted[j]['top'], predicted[j]['left']+predicted[j]['width'],

predicted[j]['top']+predicted[j]['height']]], dtype=torch.float)

x = ops.box\_iou(truth\_box, predicted\_box)

# Append possible true boxes to "filtered" array

if (x >= threshold1):

filtered.append([float(x), predicted[j]])

boxesTensors.append([predicted[j]['left'], predicted[j]['top'],predicted[j]['left']+predicted[j]['width'],predicted[j]['top']+predicted[j]['height']])

# Apply Non-maximum suppression to get 0/1 corresponding predicted box for every true box

acc=0

selected\_boxes = []

if len(filtered) > 0:

scoresTensors = tf.convert\_to\_tensor(np.array(np.array(filtered).T[0],dtype=np.float16))

boxesTensors = torch.tensor(boxesTensors)

selected\_indices = tf.image.non\_max\_suppression(boxesTensors,scoresTensors,15,threshold2)

selected\_boxes = tf.gather(boxesTensors,selected\_indices)

selected\_scores = tf.gather(scoresTensors,selected\_indices)

acc = np.sum(np.array(selected\_scores)) / len(truth)

return acc, selected\_boxes

## Boxes Recall

Boxes Recall per image is calculated as follows:

# TESTING SCENARIOS

## Single Image

Generate a random number from 0 to the images count-1 and apply the full algorithm on the randomly chosen image before testing the accuracy of the results.

***Code:***

idx=random.randint(0, len(train\_images\_all)-1)

print("Filename:",idx+1)

print("True labels:",train\_labels\_all[idx])

predicted\_boxes = rectanglesModel(train\_images\_all[idx])

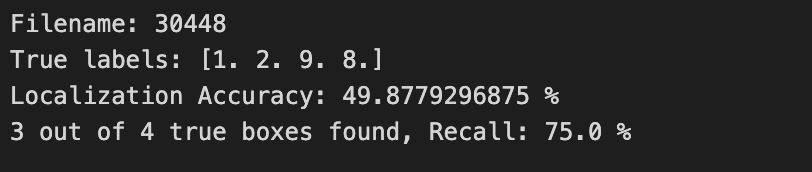
true\_boxes = train\_boxes\_all[idx]

accuracy, ret = (iouPicTest(true\_boxes,predicted\_boxes))

print("Localization Accuracy:",accuracy\*100,"%")

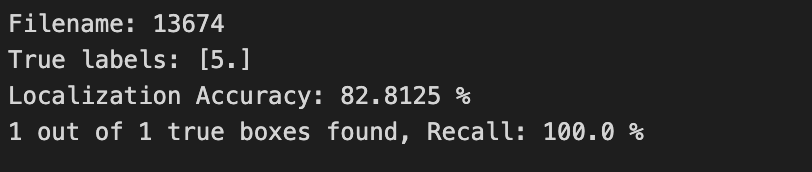
print(len(ret),"out of", len(true\_boxes), "true boxes found, Recall:", len(ret)\*100/len(true\_boxes),"%")

***Sample results from “train” images set:***



Text

Description automatically generated



## Full images folder

Taking the average accuracy of all images found in a folder.

***Code:***

print(len(train\_images\_all),"images.")

all\_acc, foundCount = getAllIOUAccuracy(train\_images\_all, train\_boxes\_all)

print("Localization Accuracy: ", all\_acc\*100, "%")

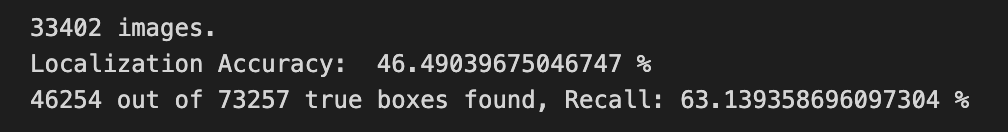
positives = len([item for sublist in train\_boxes\_all for item in sublist])

truePositives = foundCount

falseNegatives = positives - truePositives

print(truePositives,"out of", positives, "true boxes found, Recall:", truePositives\*100/positives,"%")

***Results of “train” images set:***

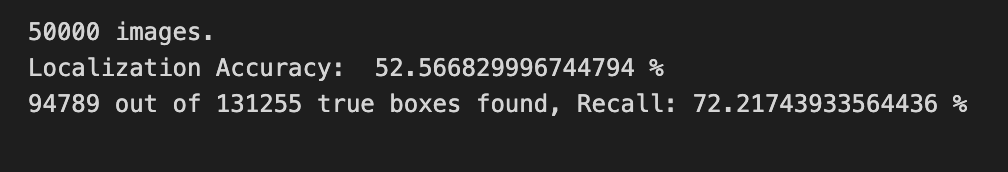
******

***Results of “test” images set:***

***Text

Description automatically generated***

***Results of 50,000 images of “extra” images set:***

******

## Image Resolution Grouping

Another key property is the image resolution, which is calculated as . Therefore, we tried splitting the images set equally to 3 groups.

Group 1 has the lowest resolution, while group 3 has the highest.

***Code:***

This function returns the indices groups of the pictures. Another function takes these indices and returns the images, labels and boxes grouped.

def split\_resolution(images, n\_groups):

image\_sizes = []

i = 0

for img in images:

image\_sizes.append([img.shape[0]\*img.shape[1], i])

i += 1

image\_sizes = sorted(image\_sizes, key=itemgetter(0))

image\_sizes = np.array(image\_sizes)

image\_sizes\_splitted = np.array\_split(image\_sizes, n\_groups)

image\_sizes\_splitted = np.array(image\_sizes\_splitted)

indices = []

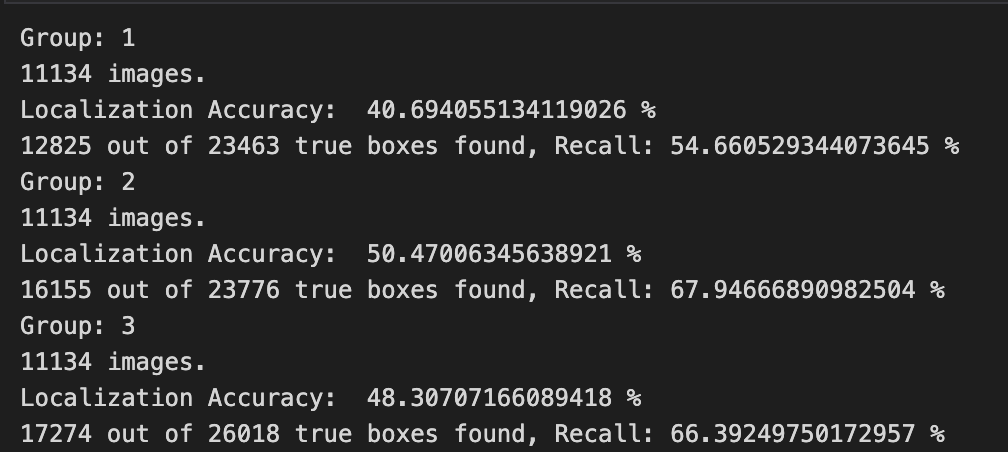
for i in range(n\_groups):

indices.append(list(image\_sizes\_splitted[i][:, 1].astype(int)))

print(len(indices[i]))

return indices

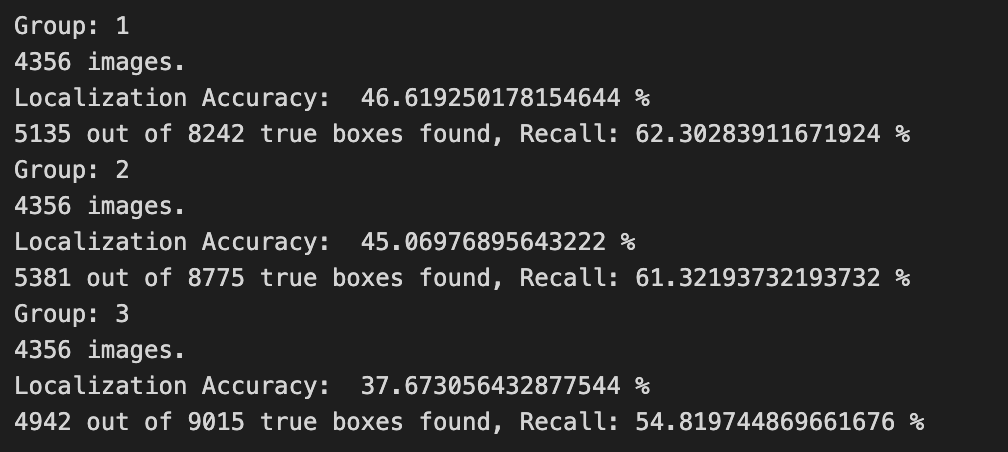
***Results of “train” images set:***

******

***Chart, bar chart

Description automatically generated***

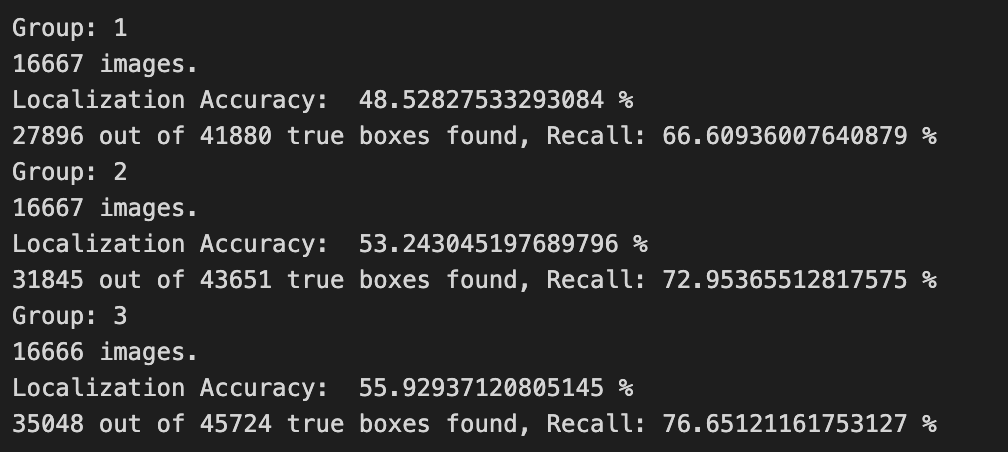
***Results of “test” images set:***

******

***Chart, bar chart

Description automatically generated***

***Results of 50,000 images of “extra” images set:***

******

***Chart, bar chart

Description automatically generated***

## Numbers-Area-to-Image-Size Ratio Grouping

Instead of testing all images at once, one shall split the images based on different properties to search for the reason behind better or worse results.

One property is the area occupied by the numbers out of the total image area. Therefore, we tried splitting the images set equally to 3 groups.

Group 1 has the lowest numbers area occupancy, while group 3 has the highest.

***Code:***

This function returns the indices groups of the pictures. Another function takes these indices and returns the images, labels and boxes grouped.

def split\_boxes\_area(images, boxes, n\_groups):

boxes\_ratio = []

i = 0

for box, img in zip(boxes, images):

sum = 0

for b in box:

sum += (b['height'] \* b['width'])

boxes\_ratio.append([sum / (img.shape[0]\*img.shape[1]), i])

i += 1

boxes\_ratio = sorted(boxes\_ratio, key=itemgetter(0))

boxes\_ratio = np.array(boxes\_ratio)

boxes\_ratio\_splitted = np.array\_split(boxes\_ratio, n\_groups)

boxes\_ratio\_splitted = np.array(boxes\_ratio\_splitted)

indices = []

for i in range(n\_groups):

indices.append(list(boxes\_ratio\_splitted[i][:, 1].astype(int)))

print(len(indices[i]))

return indices

***Results of “train” images set:***

Text

Description automatically generated

***Results of “test” images set:***

***Text

Description automatically generated***

Chart, bar chart

Description automatically generated

***Results of 50,000 images of “extra” images set:***

***Text

Description automatically generated***

Chart, bar chart

Description automatically generated

## Image Brightness Grouping

Image brightness is another key property that might affect the localization accuracy. An image might be too bright or too dark, therefore, grouping according to brightness might be insightful.

Brightness is calculated by averaging the grayscale value of an image or calculating the weighted average of the histogram of a grayscale image.

***Code:***

This function returns the indices groups of the pictures. Another function takes these indices and returns the images, labels and boxes grouped.

def split\_histogram(images, n\_groups):

image\_hists = []

i = 0

for img in images:

hist = cv2.calcHist(cv2.cvtColor(img.copy(), cv2.COLOR\_BGR2GRAY), [0], None, [256], [0, 256]).flatten()

weighted\_avg = 0

for j in range(len(hist)):

weighted\_avg += (j + 1) \* hist[j] / 256

weighted\_avg /= img.shape[0] \* img.shape[1]

image\_hists.append([weighted\_avg, i])

i+=1

image\_hists = sorted(image\_hists, key=itemgetter(0))

image\_hists = np.array(image\_hists)

image\_hists\_splitted = np.array\_split(image\_hists, n\_groups)

image\_hists\_splitted = np.array(image\_hists\_splitted)

indices = []

for i in range(n\_groups):

indices.append(list(image\_hists\_splitted[i][:, 1].astype(int)))

print(len(indices[i]))

return indices

***Results of “train” images set:***

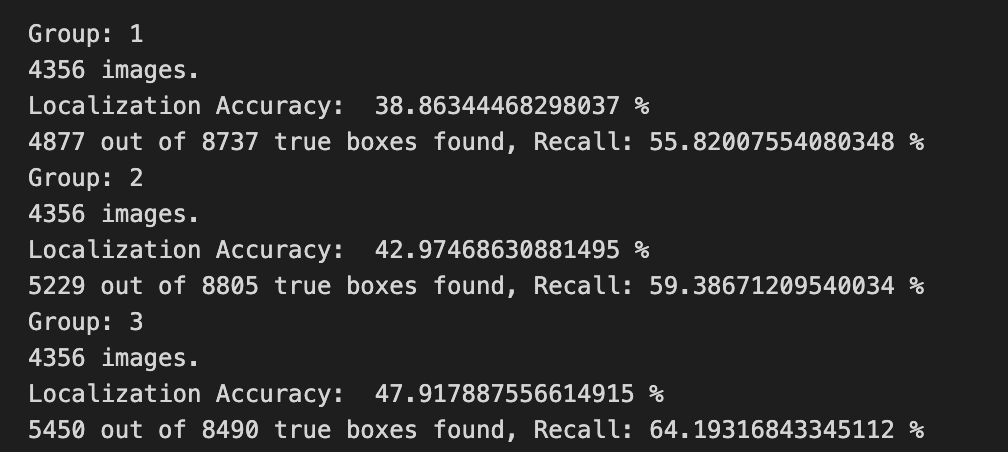
Text

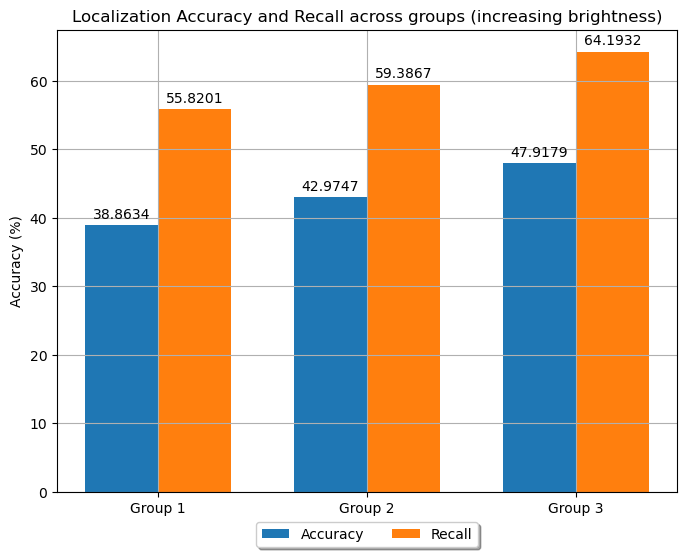
Description automatically generated

Chart, bar chart

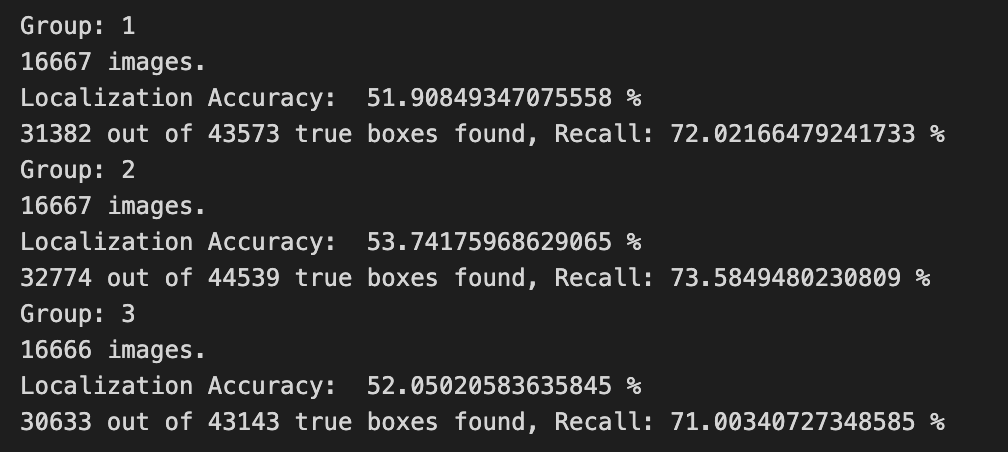
Description automatically generated

***Results of “test” images set:***





***Results of 50,000 images of “extra” images set:***



Chart, bar chart

Description automatically generated

# Source Code:

You can access the project source code on GitHub. Press [Here.](https://github.com/youssefg7/CSE483-Major-Task)