CS4053 COMPUTER VISION

ASSIGNMENT 2

FIND AND RECOGNISE BLUE SIGN OBJECTS

PROJECT REPORT

Prabhjot Kaur

18335316

Objective:

To locate and recognise the blue boxes in the given set of 33 images. We are already given dataset for the training image set.

Location: First part of this assignment is location of the sign boards in blue boxes from the 33 images using the training data which was provided in this task.

Recognition: The next part include the recognition of the data that you located.

Location:

Resize the image: As the images appearing on the screen were very large, I resized the images using

resize(image, chottu_image, Size(image.cols / 3, image.rows / 3));

This makes the visibility of the image much better than the original one on the screen to further start the procedure.

HSV Channel: I converted the image in HSV channel using

cv::Mat Gray;

cv::cvtColor(chottu_image, Gray, COLOR_BGR2HSV);

cv::split(Gray, channels);

Conversion of BGR to HSV was necessary part of this task as I only need to use the saturation channel for further tasks. Saturation channel makes it really easier to detect the blue boxes as the luminance values varies all around.

Thresholding:

After saturation it is just easier to step on Thresholding. This makes your image better in readability for computer and take less time to run your whole task. I also tried without thresholding. The major problem without thresholding was that code takes too much time to compile and then state some outputs.

threshold(S, thresholded_chottu, 20, 255,
cv::THRESH_BINARY_INVcv::THRESH_OTSU);

I did not used binary thresholding as the results were better with Ostu produced the better results here.

Edge detection:

Edge detection is an image processing technique for finding the boundaries of objects within images. It works by detecting discontinuities in brightness. Edge detection is used for image segmentation and data extraction in areas such as image processing, computer vision, and machine vision.

There were two methods of the edge detection used in open cv.

1. Laplacian of Gaussian

2. Canny edge detector

I preferred using Canny edge detection as it combines the first derivative and second derivative edge detection to compute gradient and orientation. As the priority of my task was just not to miss the edges and minimise the multiple responses to a single edge so I choose Canny edge Detector.

Remember that it takes the input of grey images so, first convert to grey scale and then proceed further.

This method accepts the following parameters −

- image − A Mat object representing the source (input image) for this operation.
- edges − A Mat object representing the destination (edges) for this operation.
- threshold1 − A variable of the type double representing the first threshold for the hysteresis procedure.
- threshold2 − A variable of the type double representing the second threshold for the hysteresis procedure.

My added code:

Canny (chottu_image, canny_edge_image_chottu, 50, 230);

Image smoothening is the inherited in the canny edge detections. As the output data is binary which is good for our finding contours.

Finding contours:

Contours are defined as the line joining all the points along the boundary of an image that are having the same intensity. Contours come handy in shape analysis, finding the size of the object of interest, and object detection.
OpenCV has findContour() function that helps in extracting the contours from the image. It works best on binary images, so we should first apply thresholding techniques, which we already had.

findContours (canny_edge_image_chottu, contours, hierarchy, RETR_EXTERNAL, cv::CHAIN_APPROX_SIMPLE);

After applying the canny edge detection as we do have everything in set of continuous points now we have to only highlight those parts of images that are provided in the training set. So only the true values should be highlighted in boxes. That symbolises that you have successfully located the objects/ blue boxes.

So, now the point is how to set the contour values so that you could locate the objects required in the task. The answer is polygons, we have to highlight only the polygons with four edges.

approxPolyDP(contours[i], approx, arcLength(contours[i], true) * 0.02, true);

As the images can be in the perspective view as well so you have to find the polygon areas and then check the rectangularity of the same. Some squares are also converted into rectangles after the perspective views of them. The rectangularity of a shape is the ratio of the region area and the area of the minimum bounding rectangle. This ratio has a maximum value of 1 for a perfect rectangle.

As we know that the boxes are rectangles and squares but it is clear from the set that the the rectangularity of the objects that we need to highlight is 0.8 (maximum overlap) in the code. Which can be also explained comparing area of contour with rot. Rectangle which all bounded it, and this rectangle had min. area that bound the contour. Taking the ratio of these two, Contours less than maximum overlap required are not taken in account.

(area / rect_area) > 0.8 (Valid contours)

Also the holes number needs to be detected and should be 1.

Further Areas of the shape should be also added to the account as unnecessary objects should be kicked out. As there are many rectangles in account we only need with the particular set of areas. In my case I tried with many different values and the best one which suits was in the range of 500-1000.

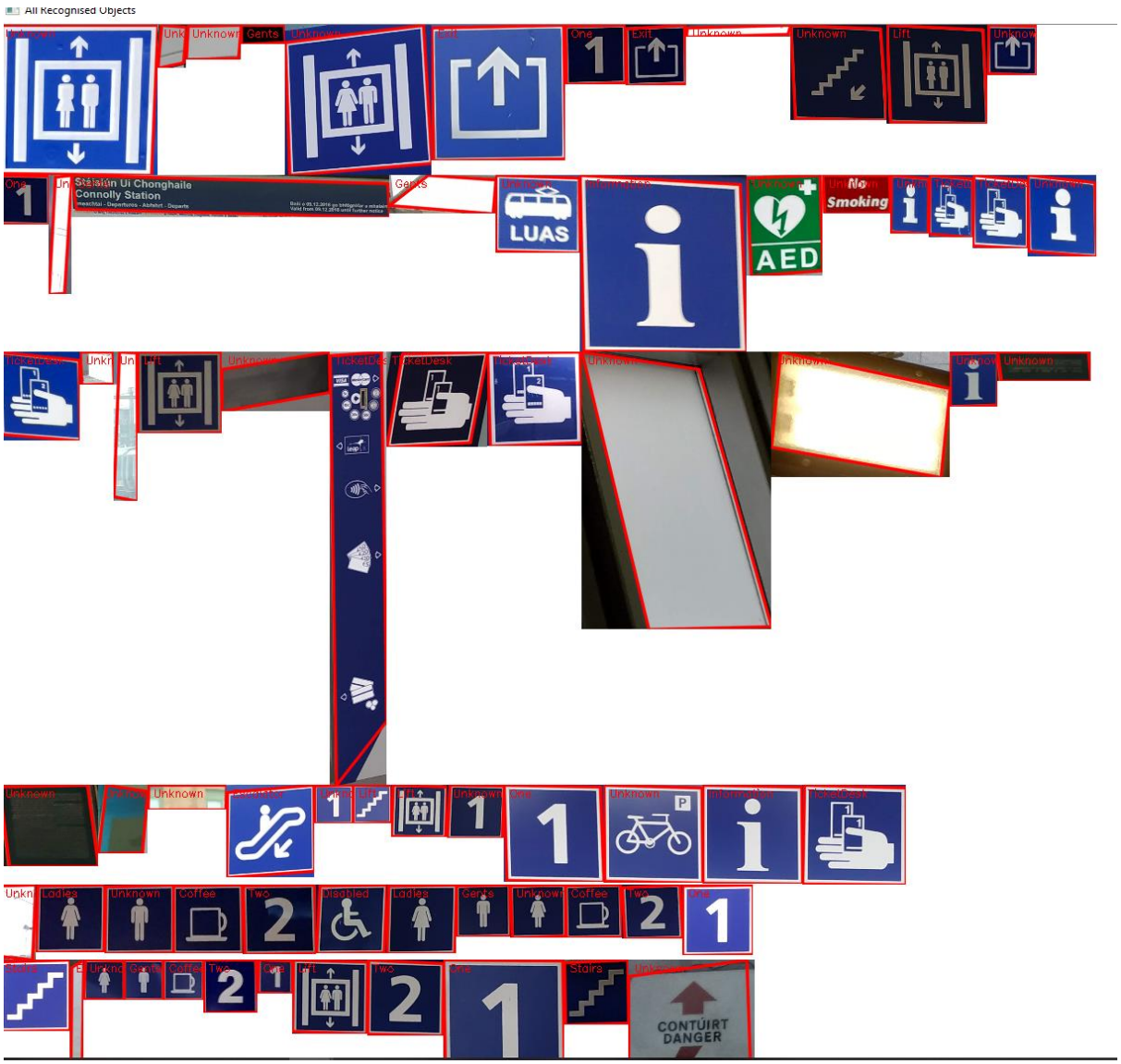
And the last condition to be taken in the account was to compare the sides of the contours. In this case you just have to be sure that the length and breadth of the shape must not be too different as some cases might not be covered in above given conditions.

So the final condition to draw a contour is :

if ((area / rect_area) > 0.8 && approx.size() == 4 && area > 1000 &&
(abs(min_rect_size.width - min_rect_size.height)) < 0.101 * large)

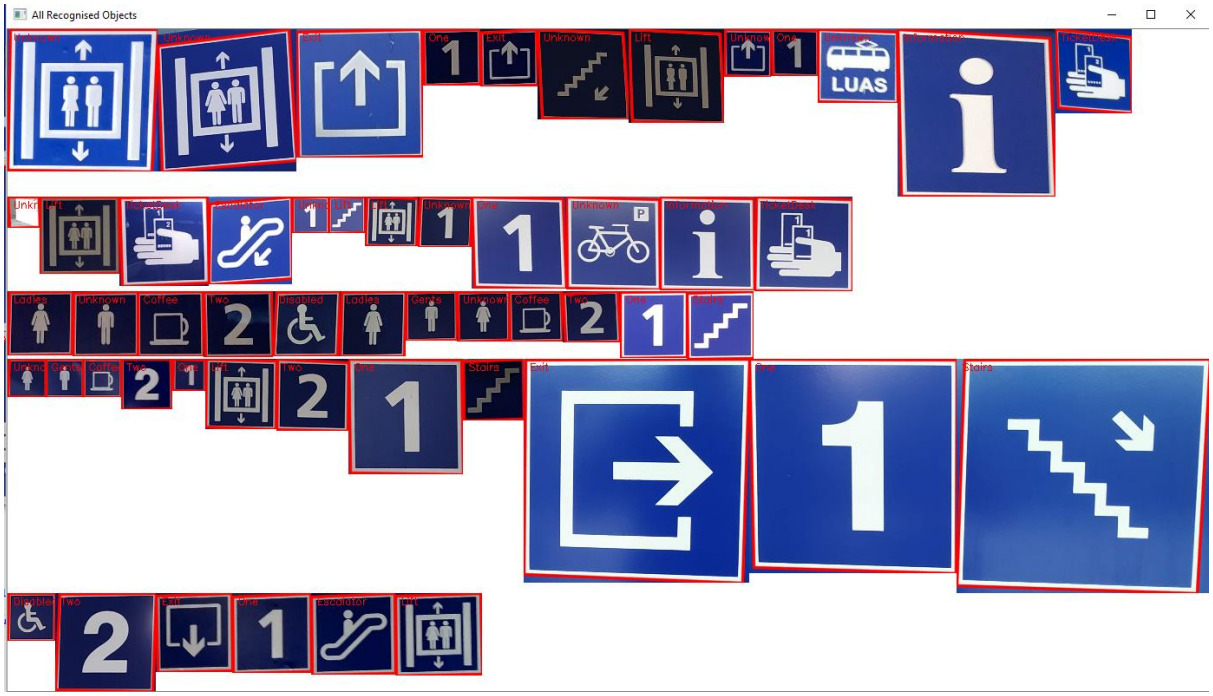
These three approximations are used in my code to locate the boxes.

I used two of these in my one approach and another one in my next approach.



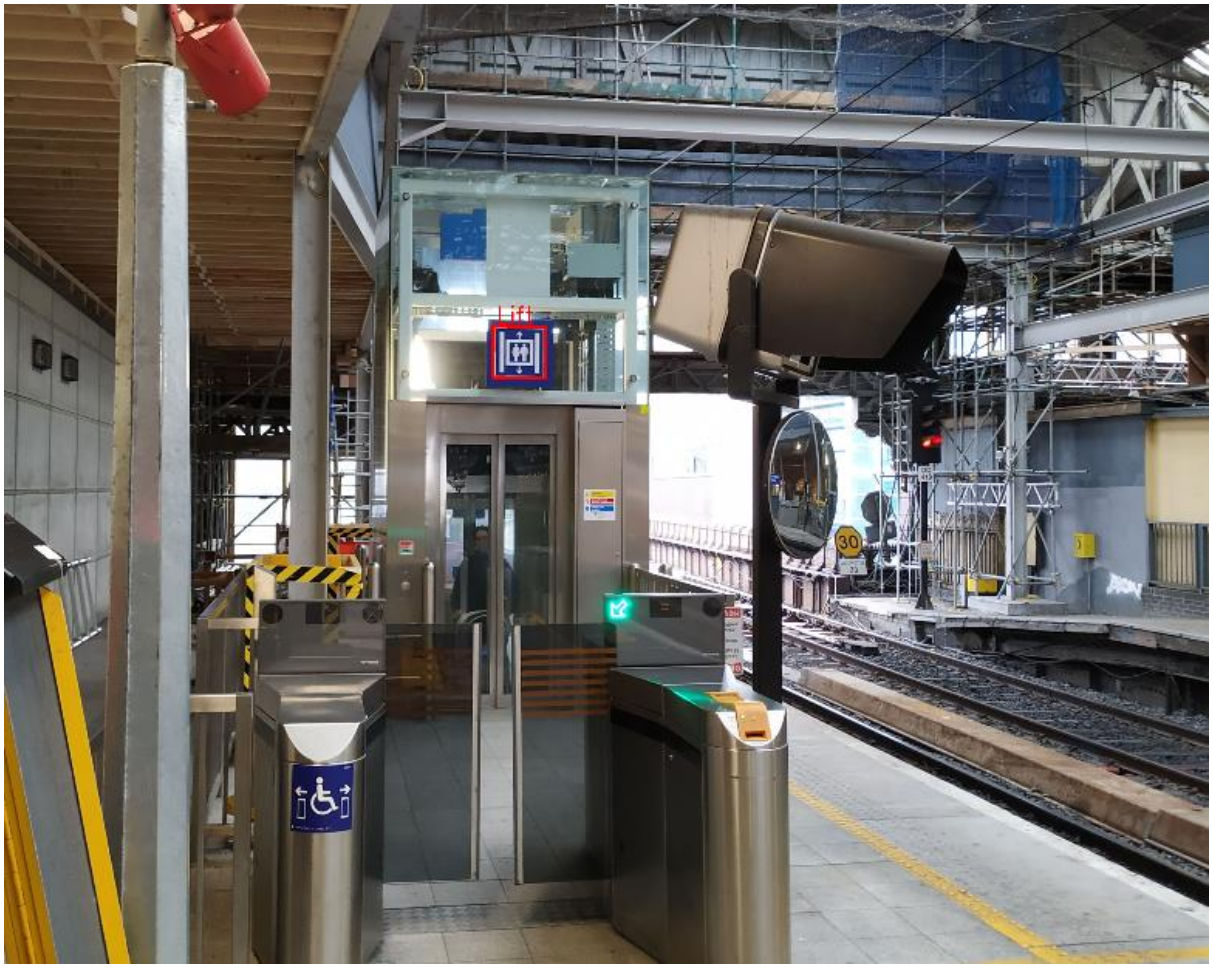
The objects located after applying the 1st two conditions without comparing the size of the length and breadth of the boxes.

As we can see that there are some boxes which are not required in our set. So, we need to get rid of this. So, we applied the third condition for comparing the size of width and height of the objects which gave better results, covering very few extra boxes.



All the located objects, with applying all the three conditions. As we can see here we do have only one extra object here.

Located objects



Here we can see that one box is located while another is not.

Cropping:

The last step in the location task was cropping up the boxes from the set of given images and add those as arguments on the addobject function, which further display all the located objects together.

cv::Rect myROI(hor_min, ver_min, hor_max - hor_min, ver_max - ver_min);

cv::Mat croppedref(origional_rect, myROI);

cv::Mat cropped;

croppedref.copyTo(cropped);

ObjectAndLocation* your_set = addObject("string", 0, 0, hor_max - hor_min, 0, hor_max - hor_min, ver_max - ver_min, 0, ver_max - ver_min, cropped);

Find the best match

Your set of cropped images has been produced and now to need to pass your set to the FindBestMatch function as arguments. Which will further compare your set to recognize the best match template for the same.

Recognition

Template Matching

Template Matching is a method for searching and finding the location of a template image in a larger image. OpenCV comes with a function cv.matchTemplate() for this purpose. It simply slides the template image over the input image (as in 2D convolution) and compares the template and patch of input image under the template image. Several comparison methods are implemented in OpenCV.It returns a grayscale image, where each pixel denotes how much does the neighbourhood of that pixel match with template.

If input image is of size (WxH) and template image is of size (wxh), output image will have a size of (W-w+1, H-h+1). Once you got the result, you can use cv.minMaxLoc() function to find where is the maximum/minimum value. Take it as the top-left corner of rectangle and take (w,h) as width and height of the rectangle. That rectangle is your region of template.

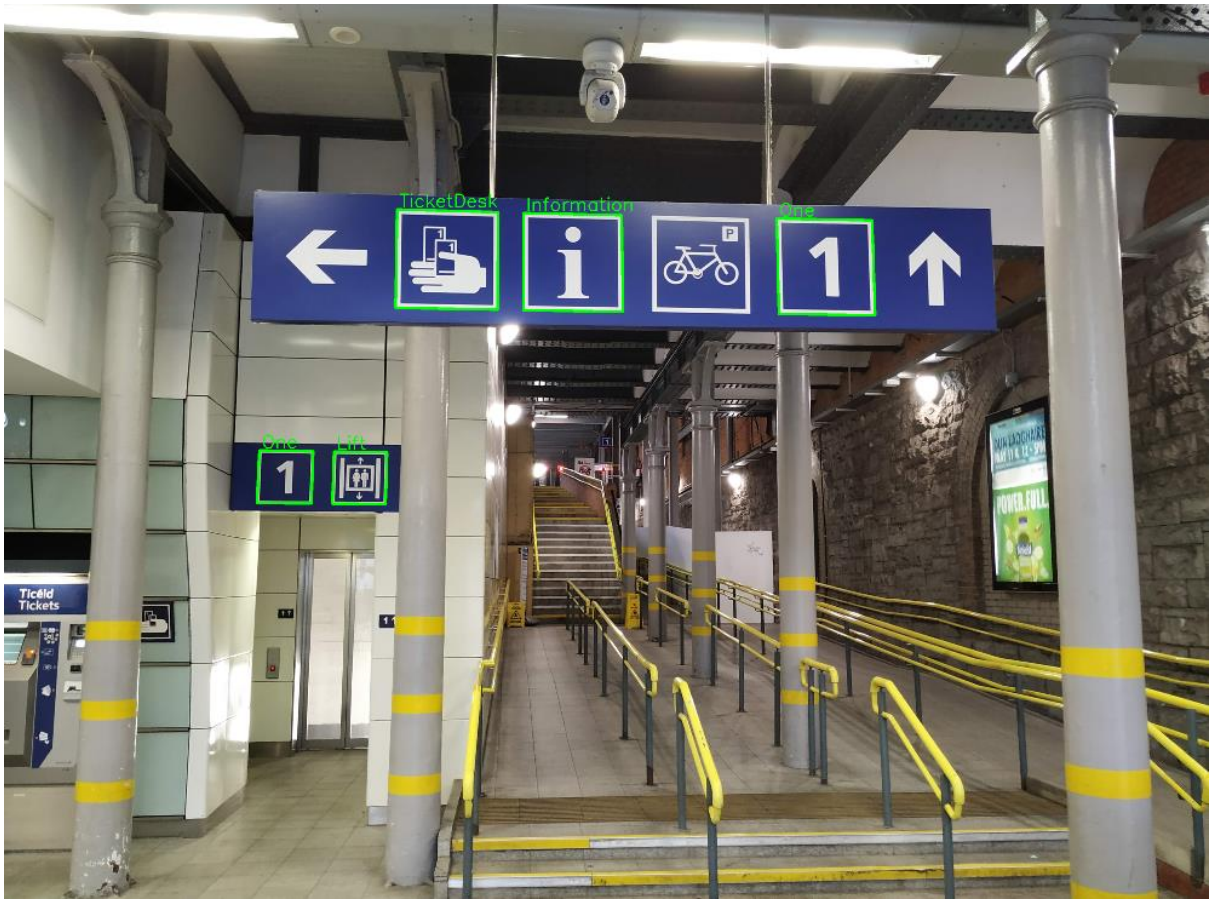
```
Mat space;

space.create(threshold_prev.cols - threshold_prev.cols + 1, threshold_prev.rows - threshold_chottu.rows + 1, CV_32FC1);

matchTemplate(threshold_prev, threshold_chottu, space, cv::TM_CCORR_NORMED);
```

Also before applying the template matching here I thresholded the images. As it is clearly visible in the images that some objects are really dark and some light. So, we couldnot set any particular threshold value for each object. So, here I used adaptive threshold.

```
adaptiveThreshold(prev1, threshold_prev, 255.0, ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY, 25, 20);
```

Some of the images after recognition are given below:

Here we can see that all the objects here are recognised, but those behind which are smaller one remains unspotted.


TicketDesk
Information
One
1
One
Lift
1
Ticéid
Tickets


TicketDesk
TicketDesk
TicketDesk
No
Smoking
AED

So, here are some positive and negative results for our implementation

RESULTS

Presision: once a object is located it is correctly recognized .

Recall: recognize percentage of the all ground truth objects

F1: balancing factor

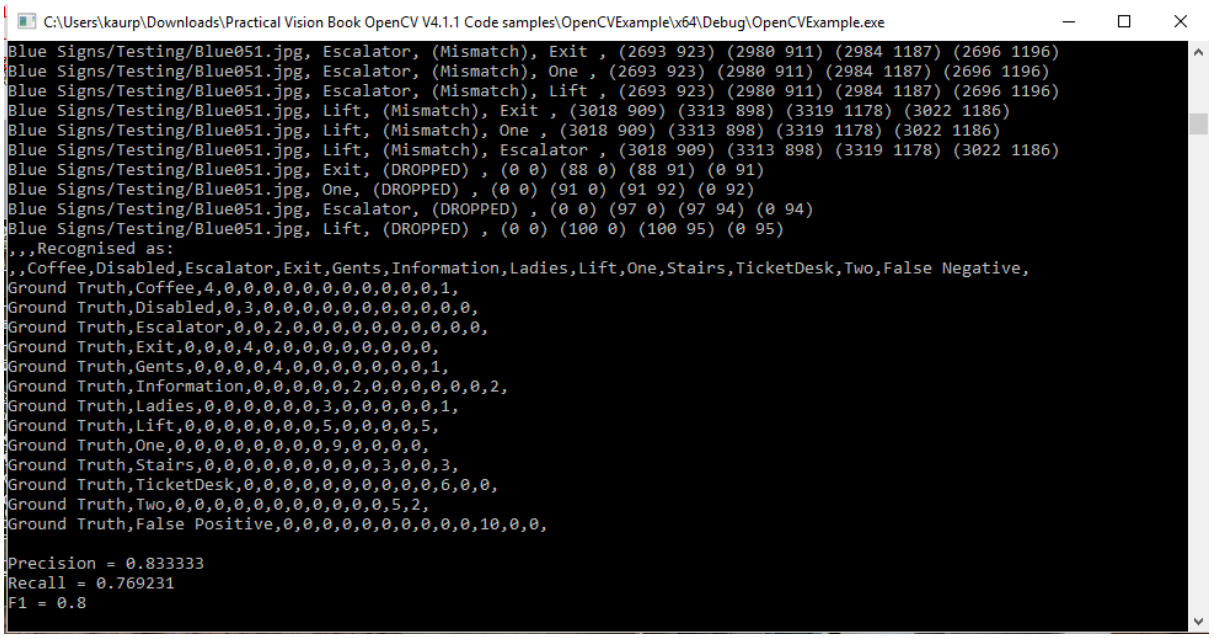
Initial Approach

Precision = 1 Recall =0.44 F1= 0.616

```
Blue Signs/Testing/Blue051.jpg, One, (Mismatch), Escalator , (2188 945) (2460 934) (2463 1203) (2191 1211)
Blue Signs/Testing/Blue051.jpg, One, (Mismatch), Lift , (2188 945) (2460 934) (2463 1203) (2191 1211)
Blue Signs/Testing/Blue051.jpg, Escalator, (Mismatch), One , (2693 923) (2980 911) (2984 1187) (2696 1196)
Blue Signs/Testing/Blue051.jpg, Escalator, (Mismatch), Lift , (2693 923) (2980 911) (2984 1187) (2696 1196)
Blue Signs/Testing/Blue051.jpg, Lift, (Mismatch), One , (3018 909) (3313 898) (3319 1178) (3022 1186)
Blue Signs/Testing/Blue051.jpg, Lift, (Mismatch), Escalator , (3018 909) (3313 898) (3319 1178) (3022 1186)
Blue Signs/Testing/Blue051.jpg, Unknown, (DROPPED) , (0 0) (85 0) (85 89) (0 89)
Blue Signs/Testing/Blue051.jpg, One, (DROPPED) , (0 0) (89 0) (89 91) (0 91)
Blue Signs/Testing/Blue051.jpg, Escalator, (DROPPED) , (0 0) (94 0) (94 92) (0 92)
Blue Signs/Testing/Blue051.jpg, Lift, (DROPPED) , (0 0) (95 0) (95 93) (0 93)
,,,Recognised as:
,,Coffee,Disabled,Escalator,Exit,Gents,Information,Ladies,Lift,One,Stairs,TicketDesk,Two,False Negative,
Ground Truth,Coffee,2,0,0,0,0,0,0,0,0,0,0,3,
Ground Truth,Disabled,0,2,0,0,0,0,0,0,0,0,0,1,
Ground Truth,Escalator,0,0,2,0,0,0,0,0,0,0,0,0,
Ground Truth,Exit,0,0,0,1,0,0,0,0,0,0,0,1,
Ground Truth,Gents,0,0,0,0,1,0,0,0,0,0,0,4,
Ground Truth,Information,0,0,0,0,0,4,0,0,0,0,0,0,4,
Ground Truth,Ladies,0,0,0,0,0,0,3,0,0,0,0,4,
Ground Truth,Lift,0,0,0,0,0,0,0,4,0,0,0,7,
Ground Truth,One,0,0,0,0,0,0,0,0,8,0,0,2,
Ground Truth,Stairs,0,0,0,0,0,0,0,0,0,2,0,7,
Ground Truth,TicketDesk,0,0,0,0,0,0,0,0,0,0,1,0,5,
Ground Truth,Two,0,0,0,0,0,0,0,0,0,0,3,3,
Ground Truth,False Positive,0,0,0,0,0,0,0,0,0,0,0,0,

Precision = 1
Recall = 0.445946
F1 = 0.616822
```

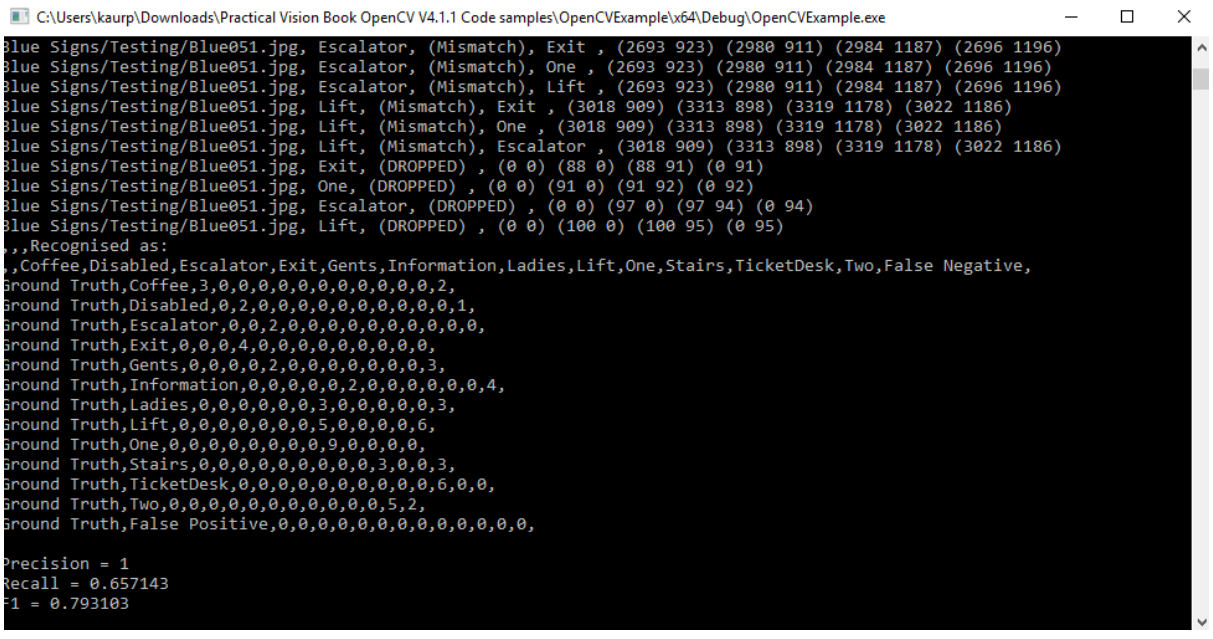
The results that came when I only applied one condition in contours part.(just checking the rectangularity)

Precision = 0.833 Recall =0.769 F1= 0.8



The result after applying first two conditions in contour part without comparing the sizes of the rectangles.

Precision = 1 Recall =0.657 F1= 0.793



The result after applying all three conditions in the contours part.

Problems and Improvements:

Recognition: After locating the boxes, some boxes were not correctly recognised. I could have applied morphology techniques such as dilation, erosion to avoid this. May be that would produce better results.

Perspective views: As some of the perspective views remain unlocated in the images. Some techniques could be used for locating. I tried using the perspective geometric transformation but that results in very weird result and many unwanted boxes were detected as well. I was unable to apply this in the code.

Canny Edge detector shortcomes: You have to choose threshold values and the width of your masks. This problem is common to all gradient based methods. Note that if you double the size of an image leaving its grey values unchanged all the gradients will be halved. This complicates the setting of any threshold. An additional problem is that the width of the mask (and hence the degree of smoothing) affects the positions of zero crossings and maximum intensity gradients in the image. The estimated position of any edge should not be affected by the size of the convolution mask.

Corners are often missed because the 1D gradient at corners is usually small. This can cause considerable difficulties for line labeling schemes because they rely on having corners and junctions being marked properly.

First derivative operators will only find step-like features. If you want to find lines you need a different operator. For example, to find bar features you could look for *peaks* (not zero crossings) from a second derivative operator. Canny did design an operator for finding lines.

Thus, differential edge detection schemes suffer both from false positives and false negatives.

Contours conditions: I could have worked on more techniques after finding contours to make sure that the we are locating the desired objects. I already applied 3 conditions but I think more conditions could be implemented for better results. As, the small boxes which were in far sights were totally missed by my approach.

Working on saturation channel: As I worked on saturation channel which also reduce some detailing of the image. I think working on other channels simultaneously would have given better results in different ways. I could have worked on all three channels differently and merged the end results all together. I think that would definitely improve the results but by low value.

Also I think more work was required in some parts to obtain better results.

References:

https://www.tutorialspoint.com/opencv/opencv_canny_edge_detection.htm

http://www.cs.tau.ac.il/~turkel/notes/can2.html

https://docs.opencv.org/master/index.html

https://docs.opencv.org/master/d4/dc6/tutorial_py_template_matching.html

A PRACTICAL INTRODUCTION TO COMPUTER VISION WITH OPENCV

Kenneth Dawson-Howe Trinity College Dublin, Ireland