CSE483: Computer Vision

Major Task-Phase 2

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

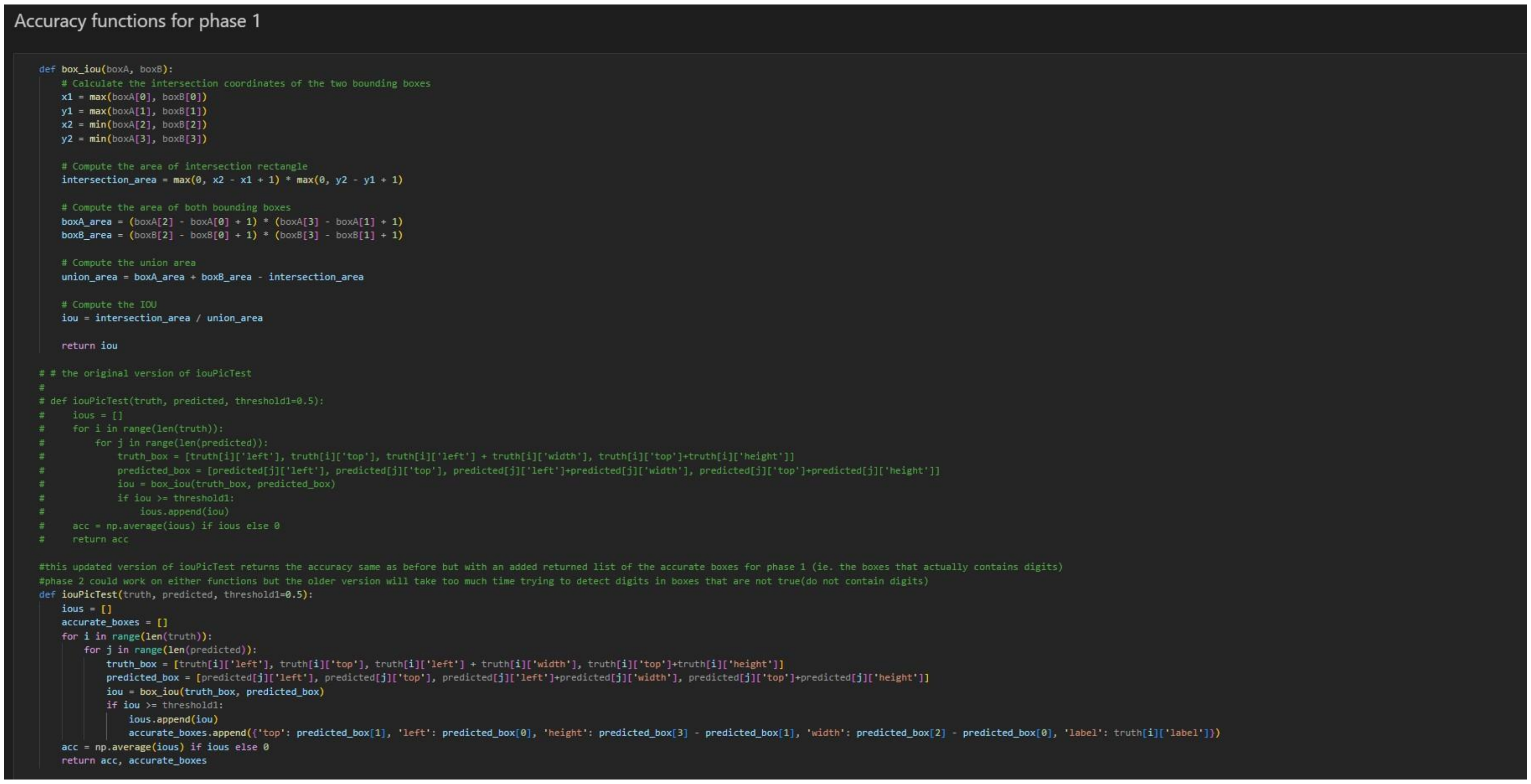
In phase 2 we need to recognize the digits in the images.

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 and give very high accuracy.

Modified accuracy calculation:


Accuracy functions for phase 1

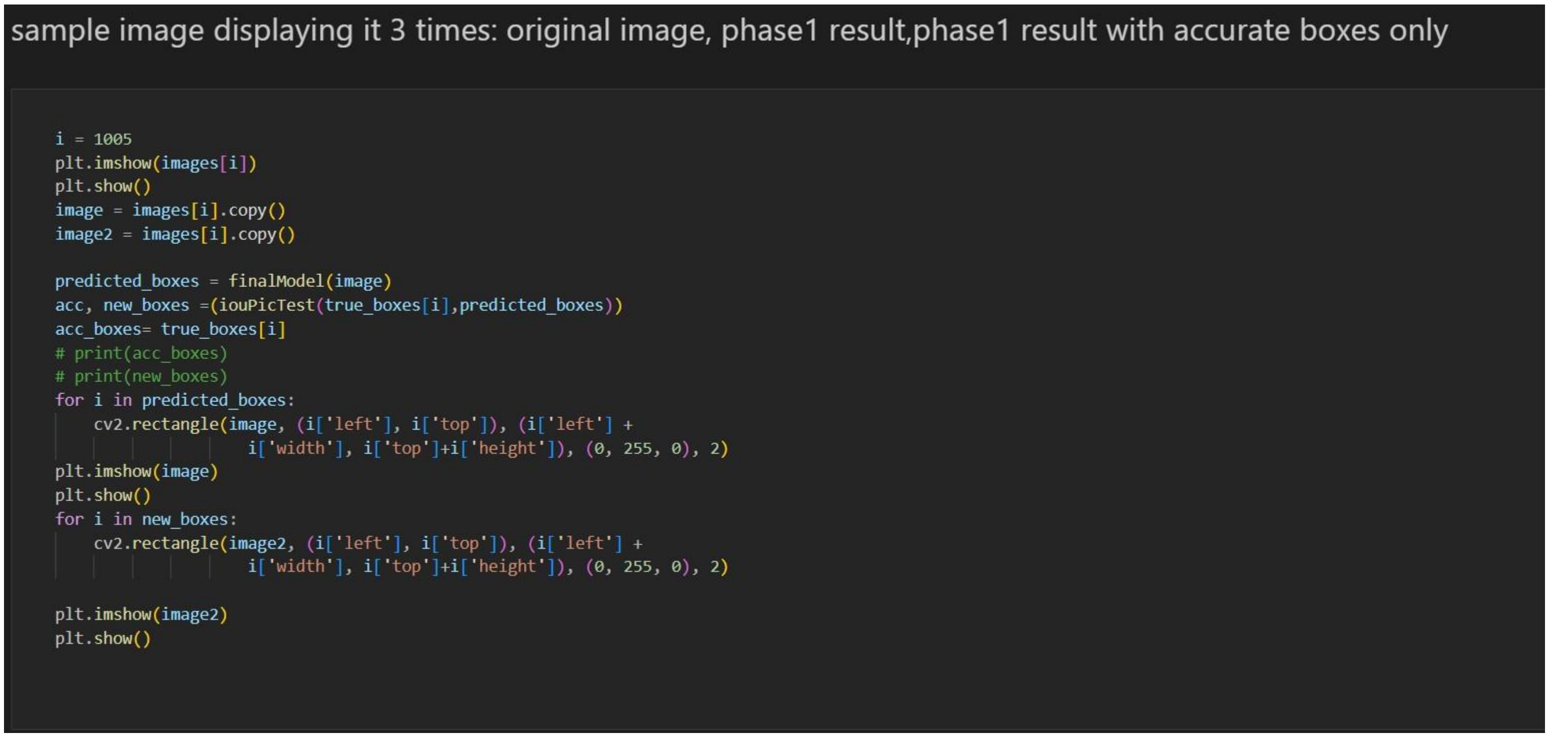
```python
def box_iou(boxA, boxB):
    # Calculate the intersection coordinates of the two bounding boxes
    x1 = max(boxA[0], boxB[0])
    y1 = max(boxA[1], boxB[1])
    x2 = min(boxA[2], boxB[2])
    y2 = min(boxA[3], boxB[3])

    # 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

# # the original version of 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

#this updated version of iouPicTest returns the accuracy same as before but with an added returned list of the accurate boxes for phase 1 (ie. the boxes that actually contains digits)
#phase 2 could work on either functions but the older version will take too much time trying to detect digits in boxes that are not true(do not contain digits)
def iouPicTest(truth, predicted, threshold1=0.5):
    ious = []
    accurate_boxes = []
    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)
                accurate_boxes.append({'top': predicted_box[1], 'left': predicted_box[0], 'height': predicted_box[3] - predicted_box[1], 'width': predicted_box[2] - predicted_box[0], 'label': truth[i]['label']})
    acc = np.average(ious) if ious else 0
    return acc, accurate_boxes
```

Figure 1 Modified accuracy calculation

This updated version of **iouPicTest** returns the accuracy same as mentioned in **Phase 1** but with an added returned list of the accurate boxes from phase 1 (i.e., the boxes that actually contains digits)

However, **Phase 2** could work on either of the two functions but the older version will take too much time trying to detect digits in boxes that are not true(i.e., The boxes that do not contain digits).

For debugging purposes, we displayed the original image, the output from phase 1, and the output of phase 1 with the accurate bounding boxes only


sample image displaying it 3 times: original image, phase1 result,phase1 result with accurate boxes only

```python
i = 1005
plt.imshow(images[i])
plt.show()
image = images[i].copy()
image2 = images[i].copy()

predicted_boxes = finalModel(image)
acc, new_boxes =(iouPicTest(true_boxes[i],predicted_boxes))
acc_boxes= true_boxes[i]
# print(acc_boxes)
# print(new_boxes)
for i in predicted_boxes:
    cv2.rectangle(image, (i['left'], i['top']), (i['left'] +
                    i['width'], i['top']+i['height']), (0, 255, 0), 2)
plt.imshow(image)
plt.show()
for i in new_boxes:
    cv2.rectangle(image2, (i['left'], i['top']), (i['left'] +
                    i['width'], i['top']+i['height']), (0, 255, 0), 2)

plt.imshow(image2)
plt.show()
```

Figure 2 Code for displaying image with correct bounding boxes only

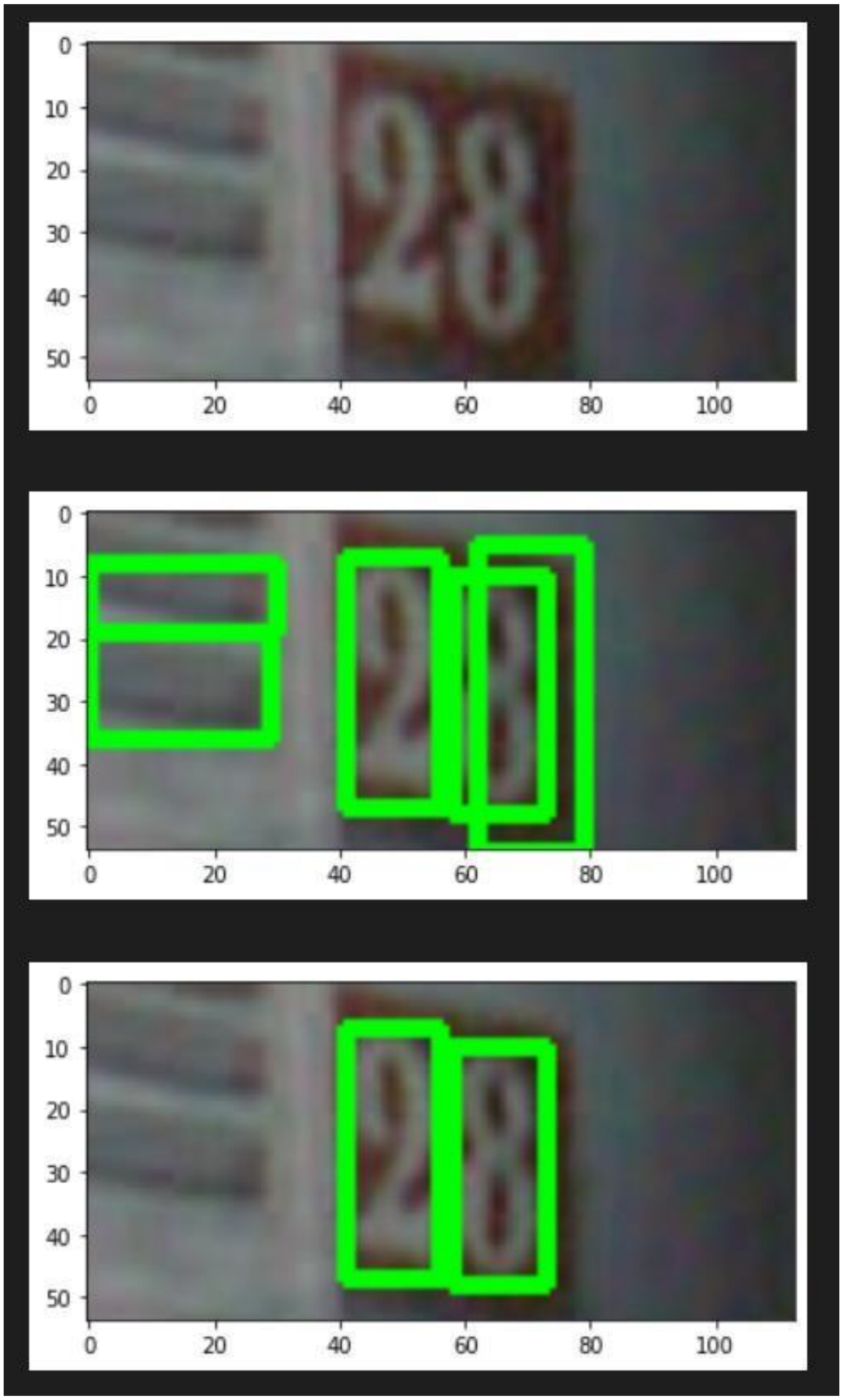
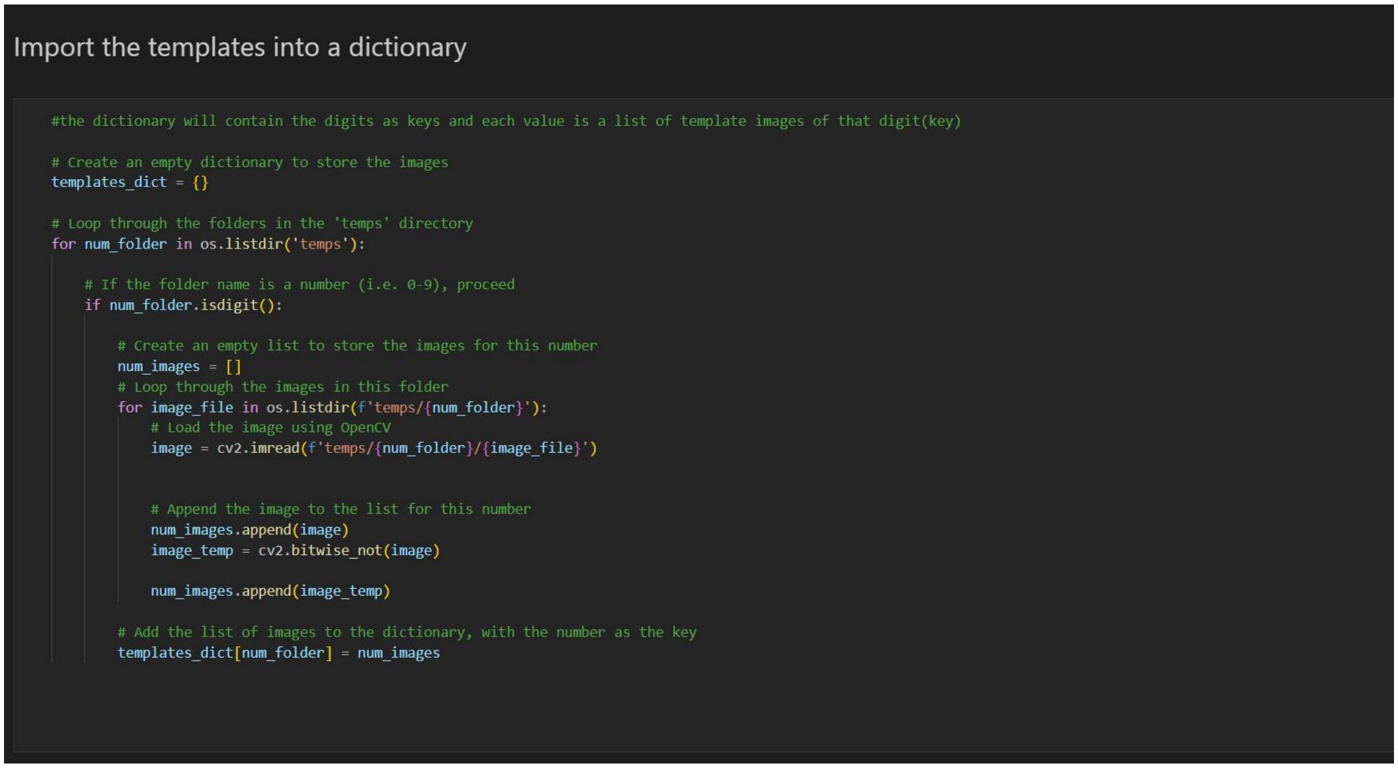


Figure 3 Original image, Phase 1 output and Phase 1 output with correct bounding boxes only

Pipeline

Step 1: Import digits template into a dictionary



Import the templates into a dictionary

```
#the dictionary will contain the digits as keys and each value is a list of template images of that digit(key)

# Create an empty dictionary to store the images
templates_dict = {}

# Loop through the folders in the 'temps' directory
for num_folder in os.listdir('temps'):

    # If the folder name is a number (i.e. 0-9), proceed
    if num_folder.isdigit():

        # Create an empty list to store the images for this number
        num_images = []
        # Loop through the images in this folder
        for image_file in os.listdir(f'temps/{num_folder}'):
            # Load the image using OpenCV
            image = cv2.imread(f'temps/{num_folder}/{image_file}')


            # Append the image to the list for this number
            num_images.append(image)
            image_temp = cv2.bitwise_not(image)

            num_images.append(image_temp)

        # Add the list of images to the dictionary, with the number as the key
        templates_dict[num_folder] = num_images
```

Figure 4 Import digits template

1. Create an empty dictionary **"templates_dict"** to store the template images of digits. The keys of the dictionary are the digits 0-9 and the values are lists of template images for each digit.
2. Loop through the folders in the **"temps"** directory and check if the folder name is a digit. If it is a digit, an empty list **"num_images"** is created to store the images for that digit.
3. Loop through the images in that folder, loads each image using OpenCV **cv2.imread()** function
4. Append it to the **"num_images"** list.
5. Creates a new image template by applying a bitwise not operation to the original image and appends it to the **"num_images"** list.(To have digits in black and background in white for example)
6. Adds the list of images for the current digit to the **"templates_dict"** dictionary, with the digit as the key.

Template images for digits

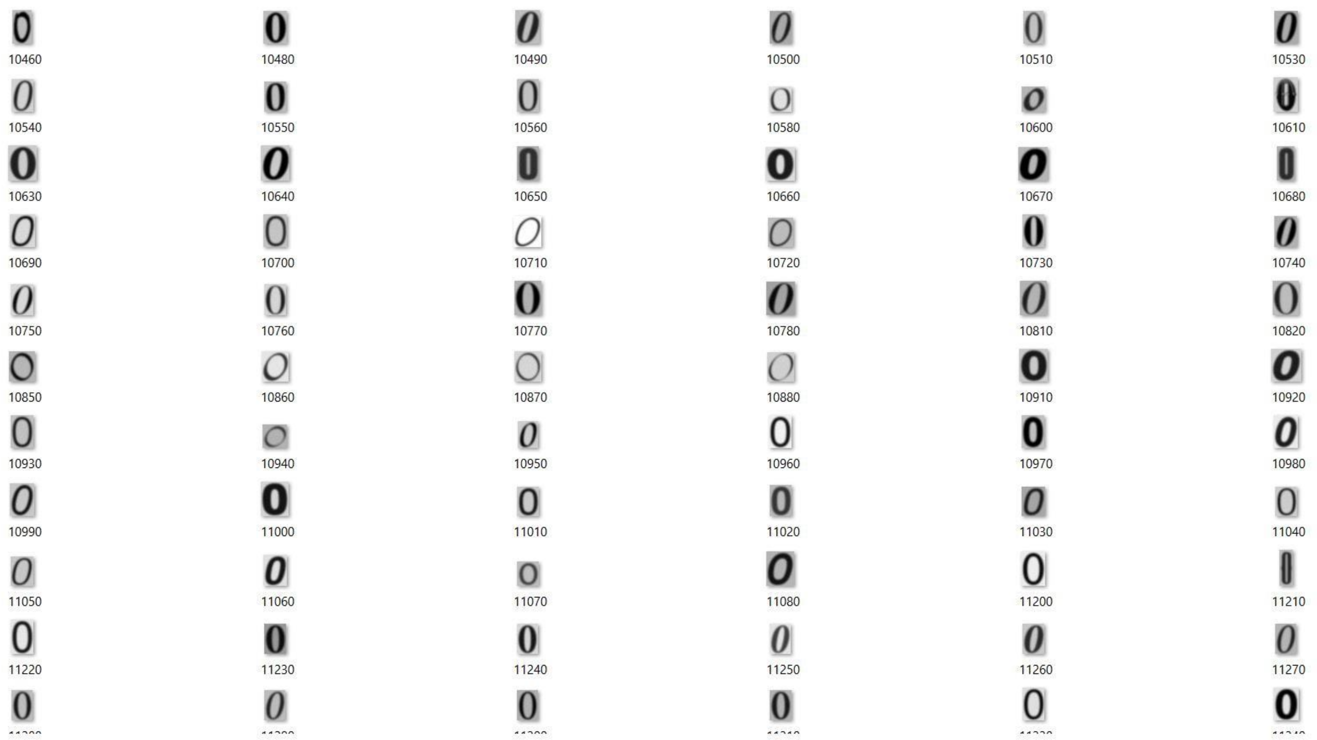


Figure 5 Template images for digit 0

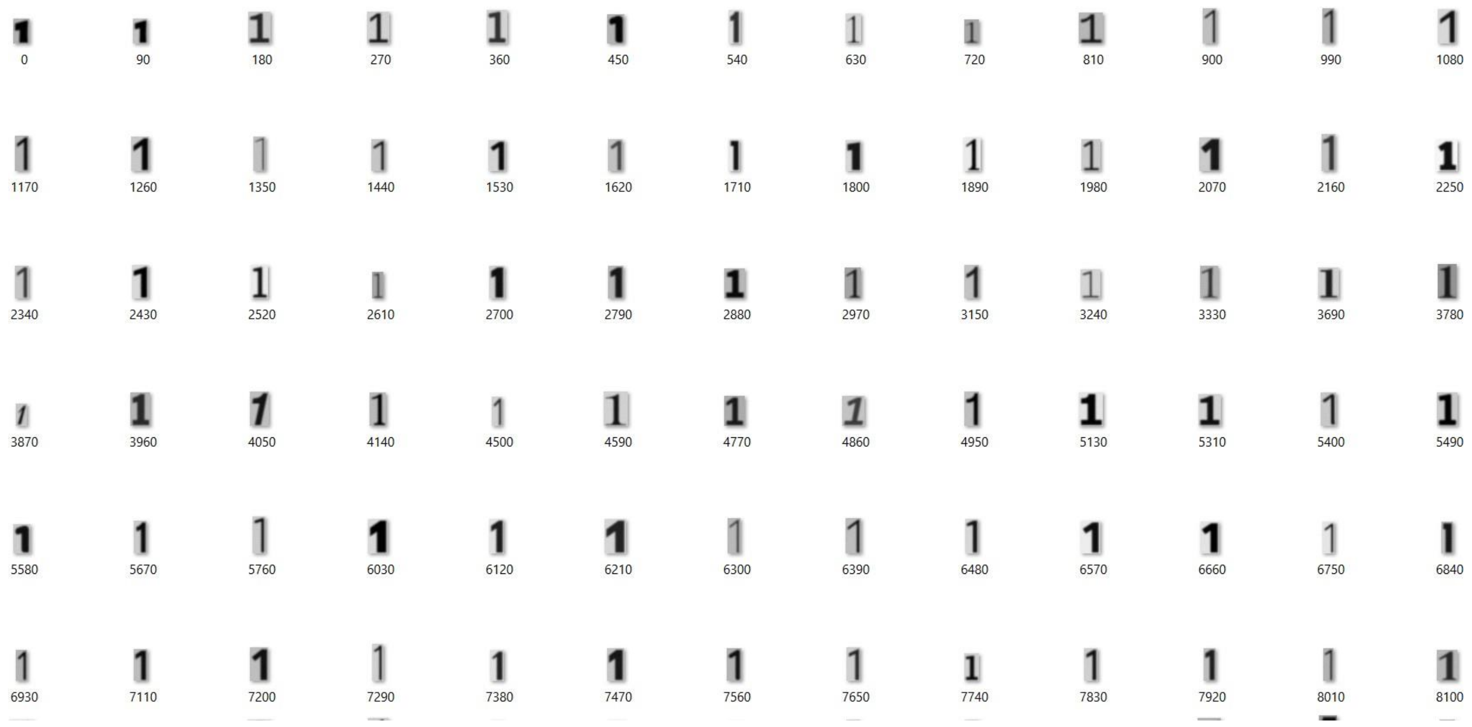


Figure 6 Template images for digit 1

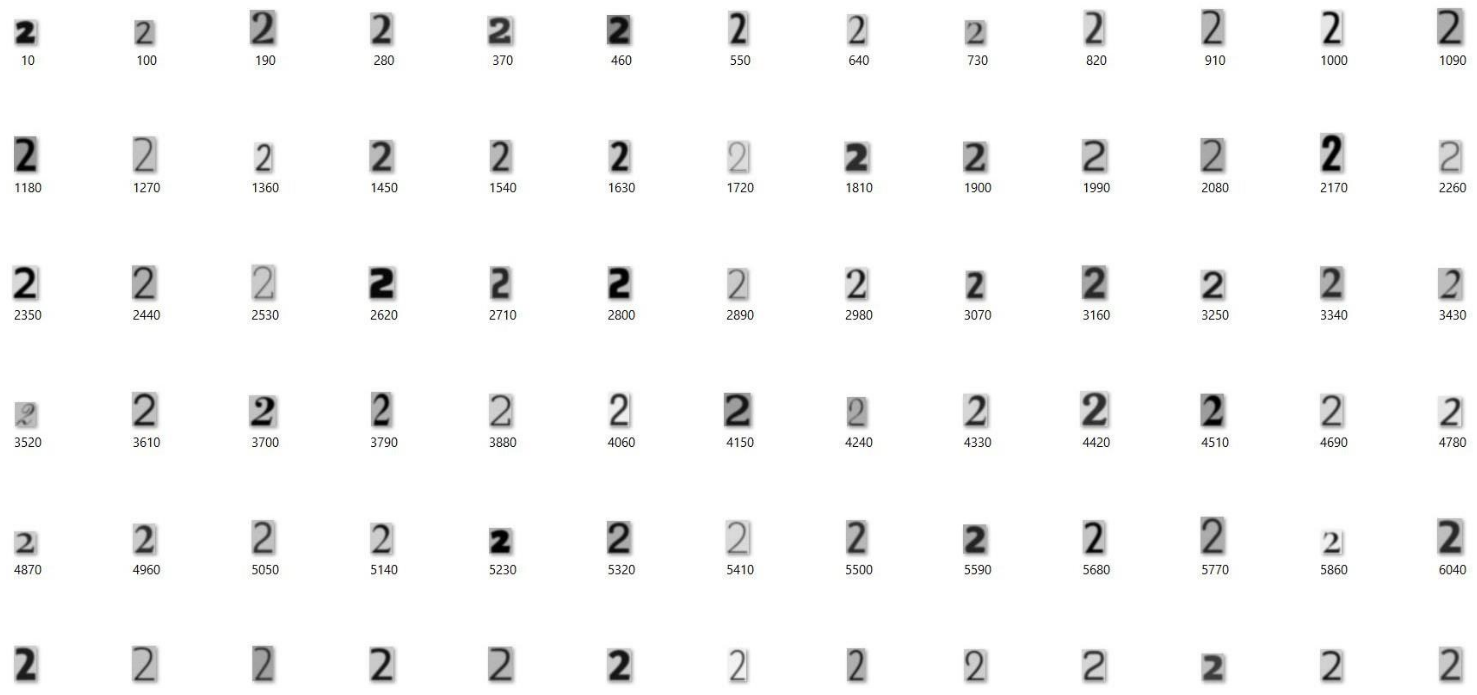


Figure 7 Template images for digit 2

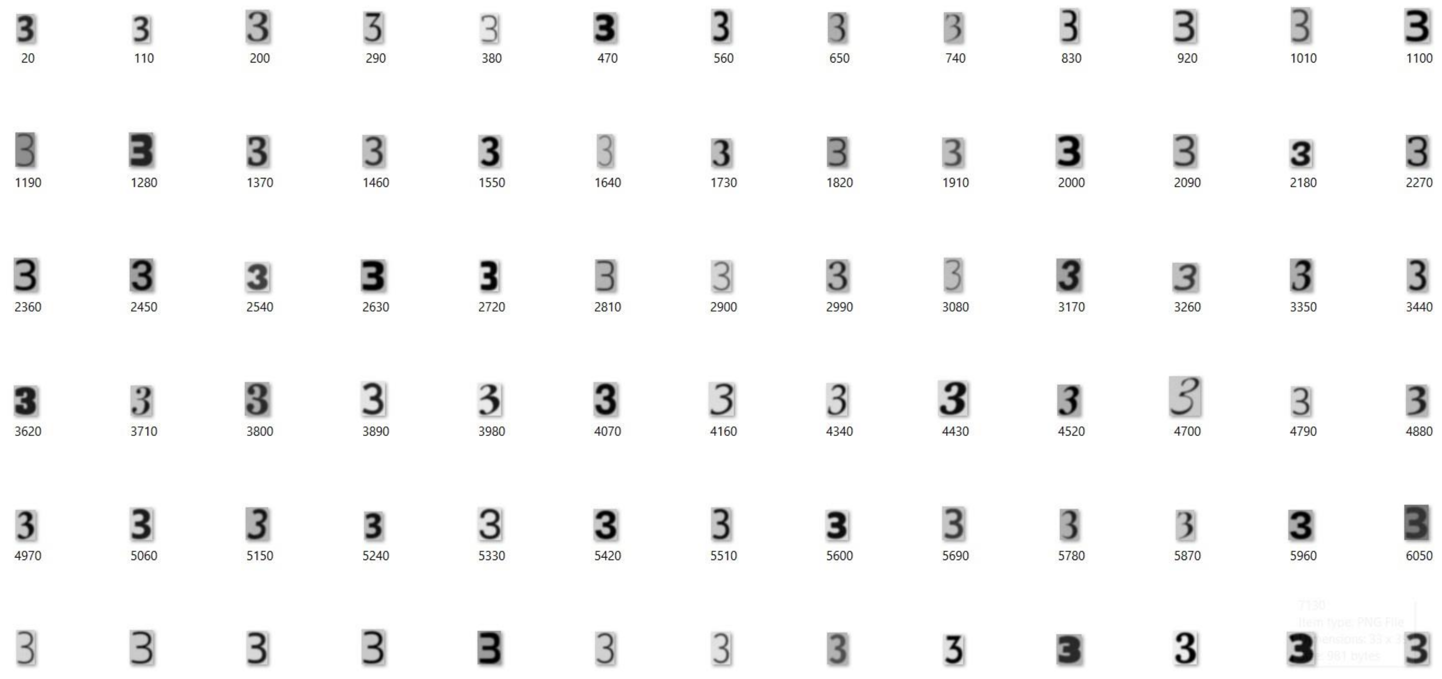


Figure 8 Template images for digit 3

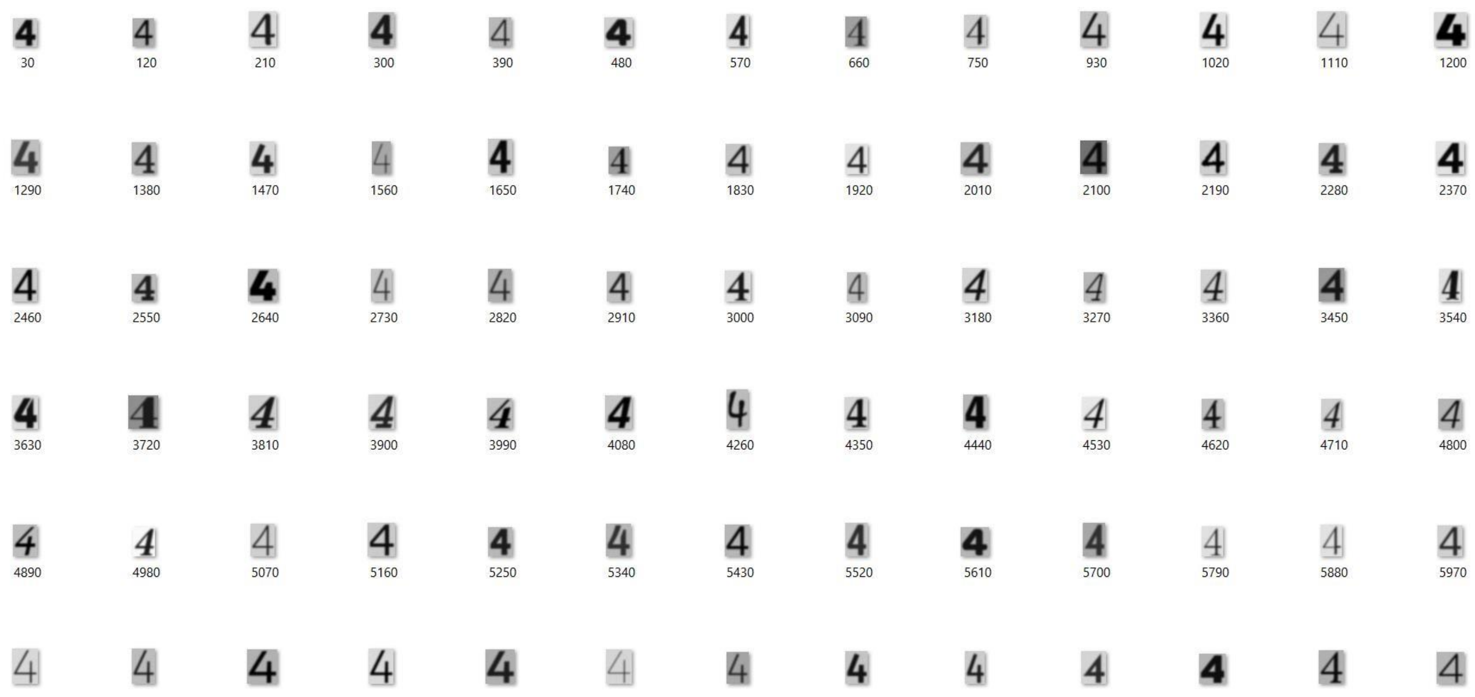


Figure 9 Template images for digit 4

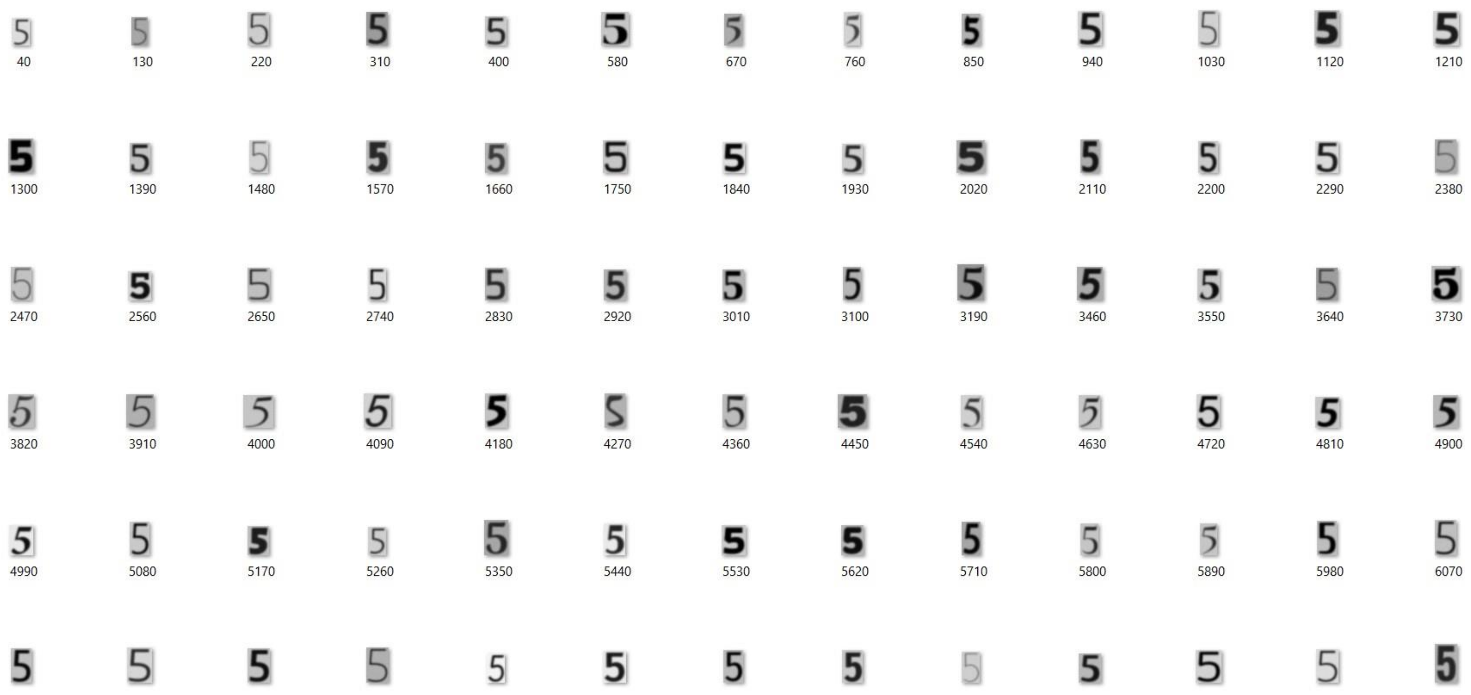


Figure 10 Template images for digit 5

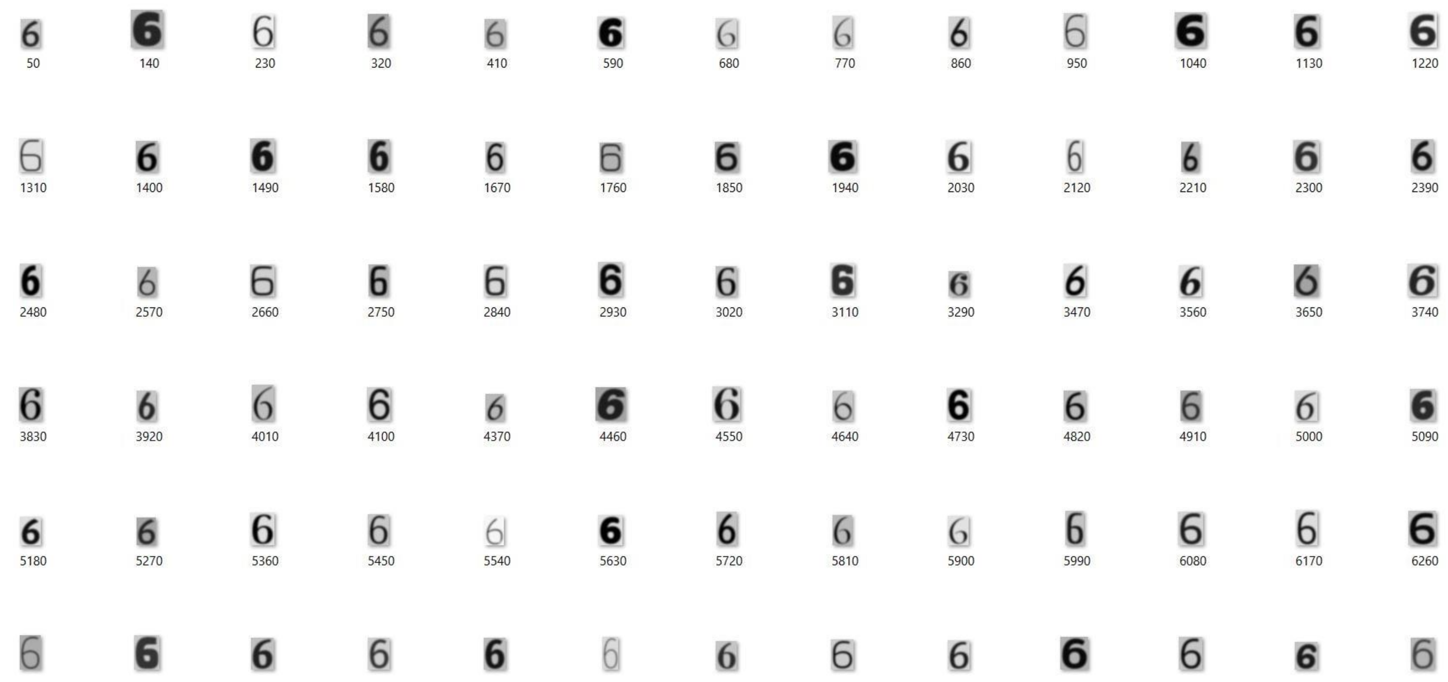


Figure 11 Template images for digit 6

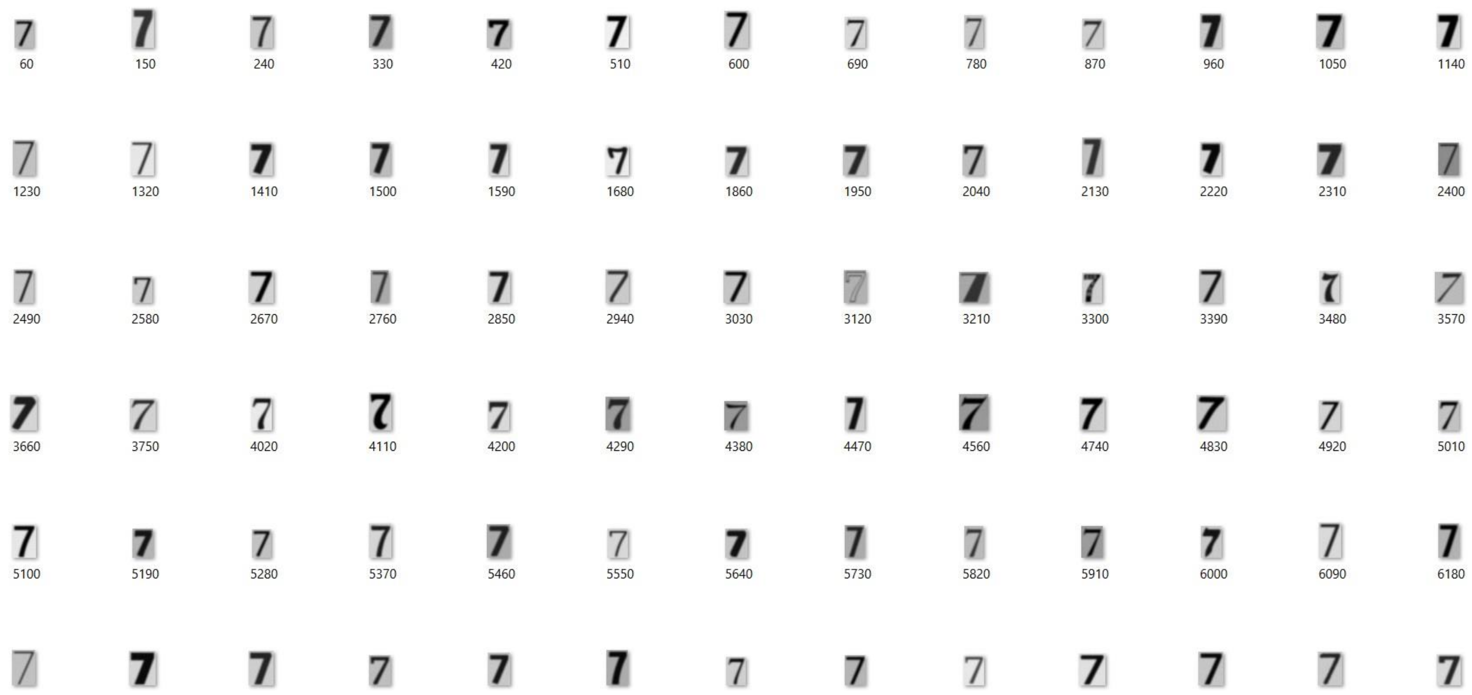


Figure 12 Template images for digit 7

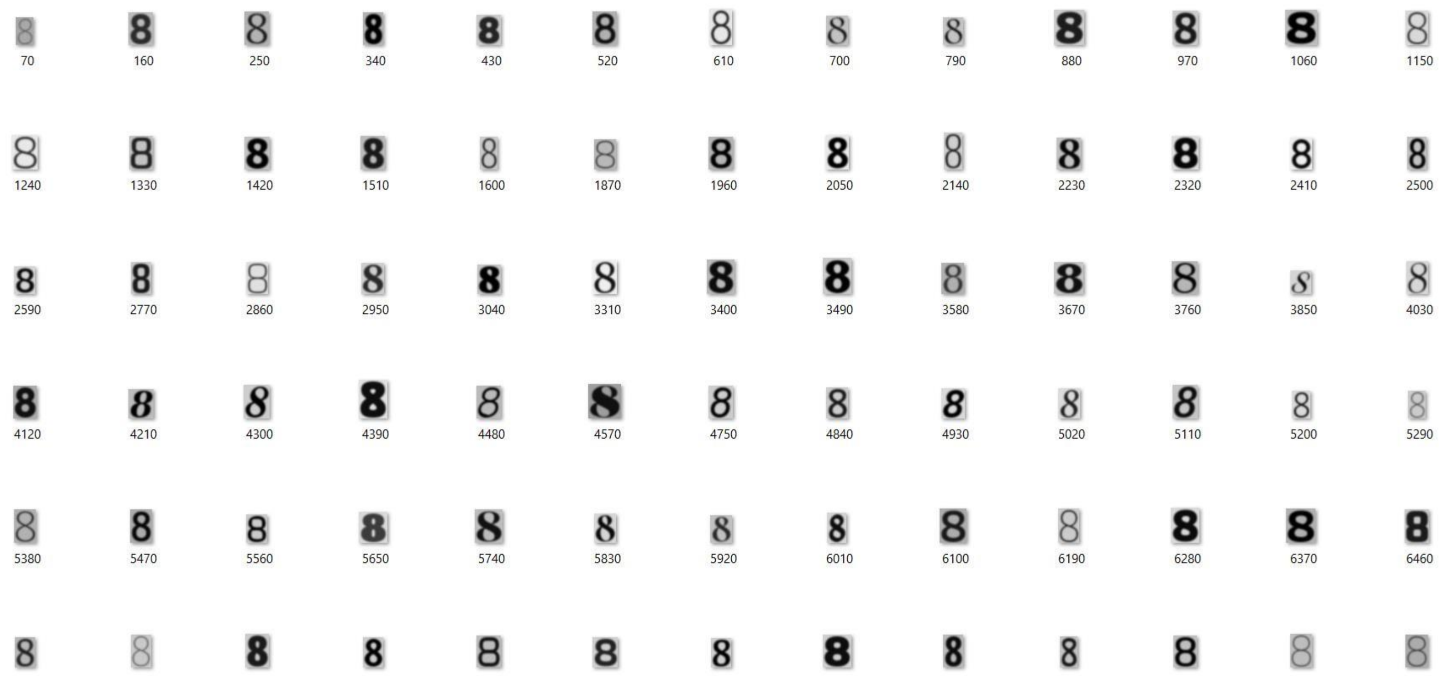


Figure 13 Template images for digit 8

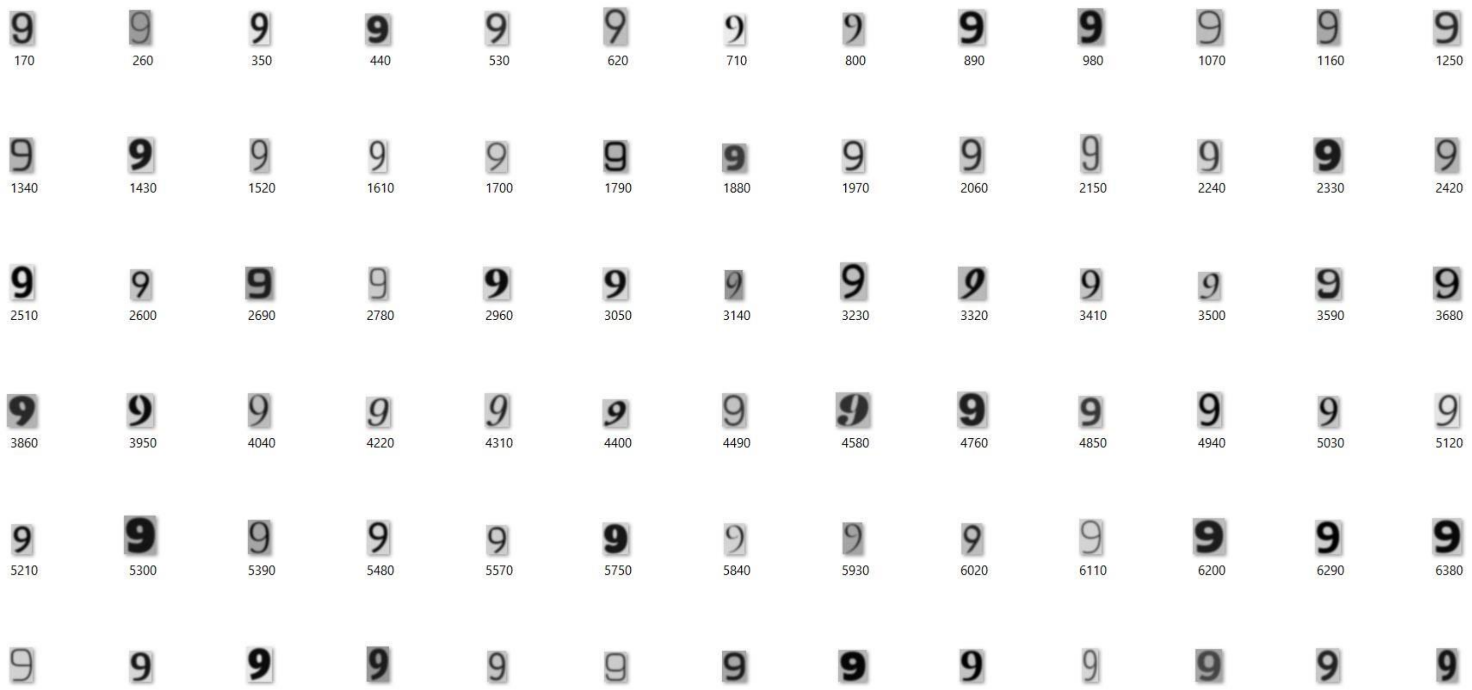
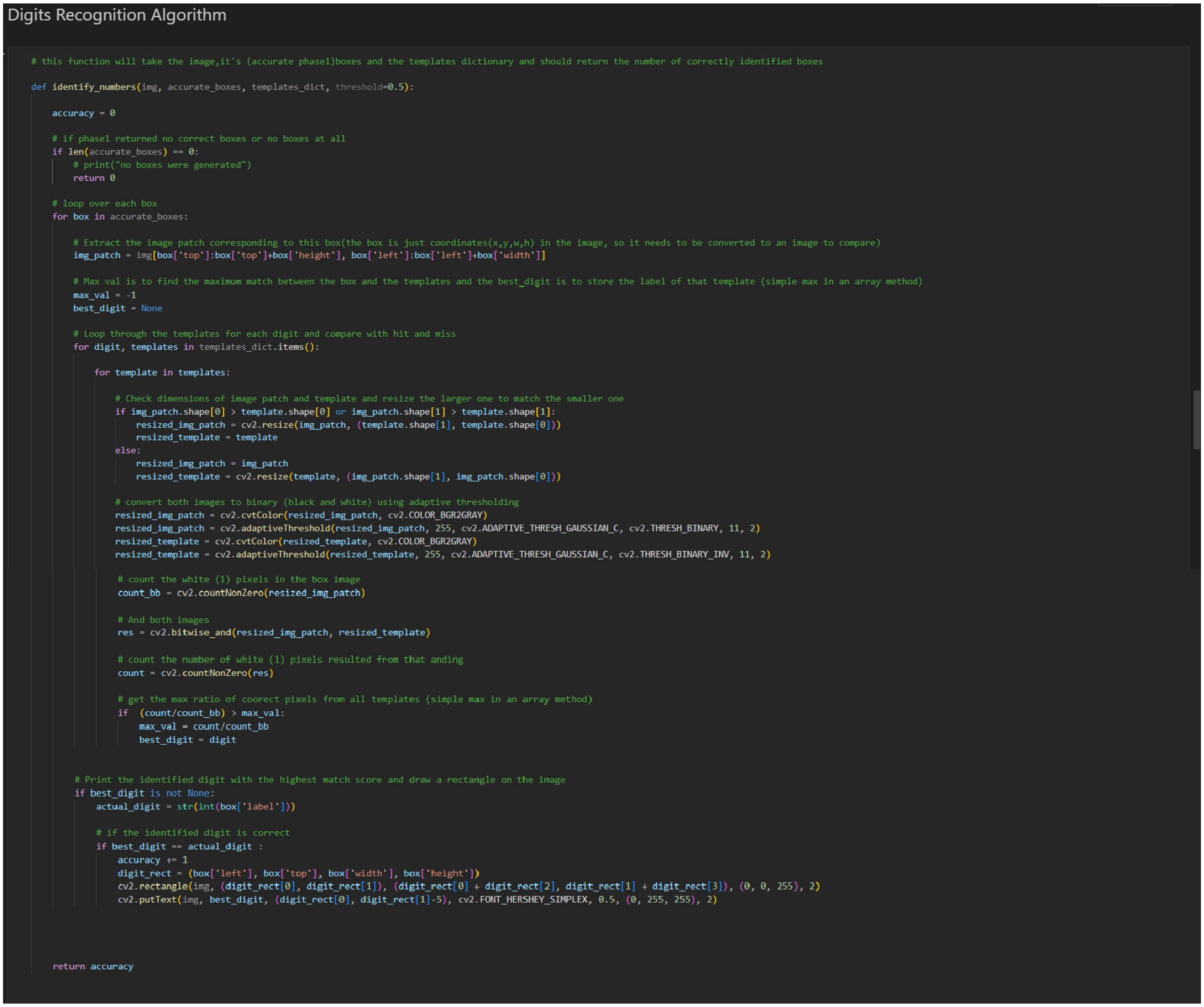


Figure 14 Template images for digit 9

Step 2: Implement the digits recognition function



Digits Recognition Algorithm

```python
# this function will take the image,it's (accurate phase1)boxes and the templates dictionary and should return the number of correctly identified boxes

def identify_numbers(img, accurate_boxes, templates_dict, threshold=0.5):

    accuracy = 0

    # if phase1 returned no correct boxes or no boxes at all
    if len(accurate_boxes) == 0:
        # print("no boxes were generated")
        return 0

    # loop over each box
    for box in accurate_boxes:

        # Extract the image patch corresponding to this box(the box is just coordinates(x,y,w,h) in the image, so it needs to be converted to an image to compare)
        img_patch = img[box['top']:box['top']+box['height'], box['left']:box['left']+box['width']]

        # Max val is to find the maximum match between the box and the templates and the best_digit is to store the label of that template (simple max in an array method)
        max_val = -1
        best_digit = None

        # Loop through the templates for each digit and compare with hit and miss
        for digit, templates in templates_dict.items():

            for template in templates:

                # Check dimensions of image patch and template and resize the larger one to match the smaller one
                if img_patch.shape[0] > template.shape[0] or img_patch.shape[1] > template.shape[1]:
                    resized_img_patch = cv2.resize(img_patch, (template.shape[1], template.shape[0]))
                    resized_template = template
                else:
                    resized_img_patch = img_patch
                    resized_template = cv2.resize(template, (img_patch.shape[1], img_patch.shape[0]))

                # convert both images to binary (black and white) using adaptive thresholding
                resized_img_patch = cv2.cvtColor(resized_img_patch, cv2.COLOR_BGR2GRAY)
                resized_img_patch = cv2.adaptiveThreshold(resized_img_patch, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
                resized_template = cv2.cvtColor(resized_template, cv2.COLOR_BGR2GRAY)
                resized_template = cv2.adaptiveThreshold(resized_template, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 11, 2)

                # count the white (1) pixels in the box image
                count_bb = cv2.countNonZero(resized_img_patch)

                # And both images
                res = cv2.bitwise_and(resized_img_patch, resized_template)

                # count the number of white (1) pixels resulted from that anding
                count = cv2.countNonZero(res)

                # get the max ratio of coorect pixels from all templates (simple max in an array method)
                if  (count/count_bb) > max_val:
                    max_val = count/count_bb
                    best_digit = digit


        # Print the identified digit with the highest match score and draw a rectangle on the image
        if best_digit is not None:
            actual_digit = str(int(box['label']))

            # if the identified digit is correct
            if best_digit == actual_digit :
                accuracy += 1
                digit_rect = (box['left'], box['top'], box['width'], box['height'])
                cv2.rectangle(img, (digit_rect[0], digit_rect[1]), (digit_rect[0] + digit_rect[2], digit_rect[1] + digit_rect[3]), (0, 0, 255), 2)
                cv2.putText(img, best_digit, (digit_rect[0], digit_rect[1]-5), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 255), 2)



    return accuracy
```

Figure 15 Digit recognition function

The function **identify_numbers** take as input an image (**img**), a list of accurate bounding boxes for digits in the image (**"accurate_boxes"**) from Phase 1 algorithm , the dictionary of digit templates (**"templates_dict"**) created in step 1, and a threshold value with default value=0.5 (**"threshold"**).

1- Initializing a variable **"accuracy"** to 0. This variable will be used to keep track of the number of correctly identified digits.

2- Check if there are any accurate bounding boxes generated by the phase 1 or if there are any boxes at all. If there are no accurate boxes generated by phase 1 or there are no boxes at all, the function returns 0.

3- Otherwise, the function loops over each bounding box in the **"accurate_boxes"** list.

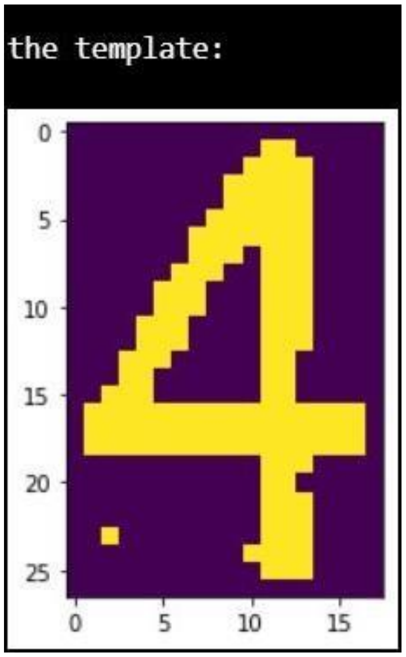
4- For each box:

    a. extract the corresponding image patch from the input image by slicing the input image using the top, left, height, and width attributes of the box.

    b. Loop over each digit in the **"templates_dict"** dictionary and each template image for that digit.

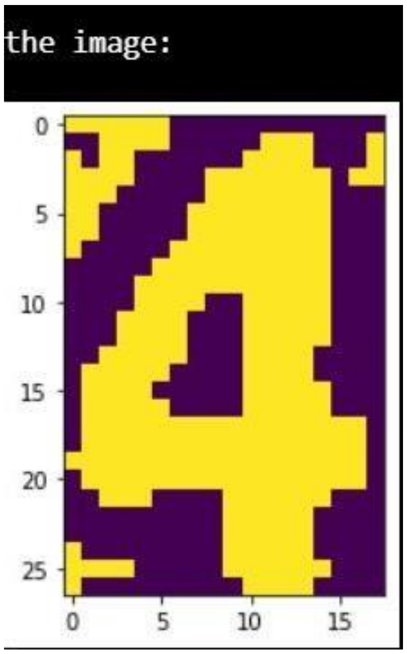
    c. For each template, check if the dimensions of the current image patch and template are different. If they are ,resize the image patch and template to the same dimensions and convert them to black and white using adaptive thresholding.

    d. Then ,compute the ratio of white pixels in the image patch that also appear in the template by first counting the number of white pixels in the resized image patch, then perform a **bitwise AND** operation between the resized image patch and the resized template. Then count the number of white pixels in the result image, which represents the number of pixels in the image patch that also appear in the template



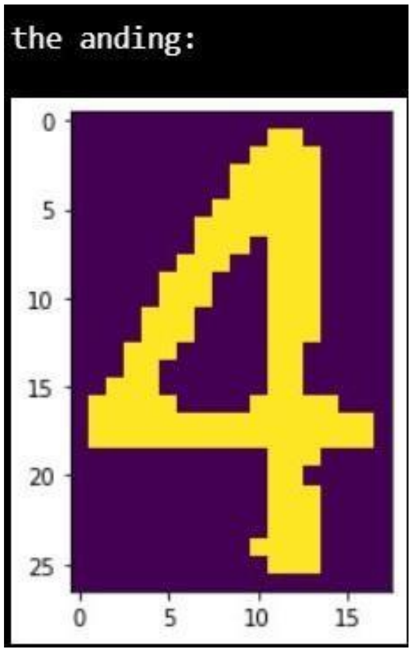
```
the template:
```

Figure 21 Template 1 image



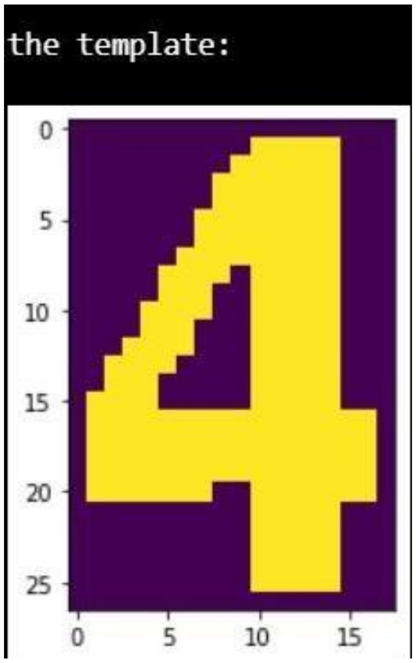
```
the image:
```

Figure 20 Original Image



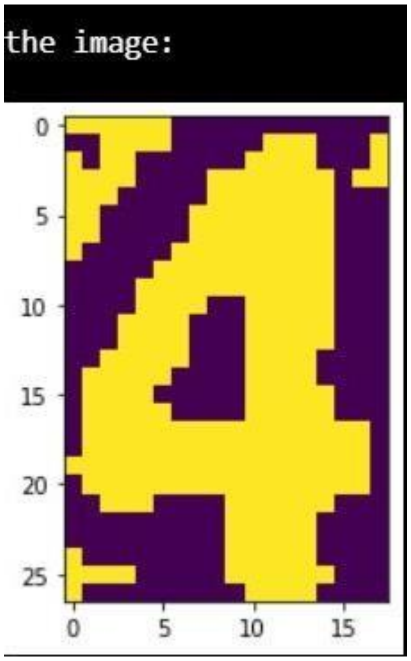
```
the anding:
```

Figure 19 ANDing between them(T1)



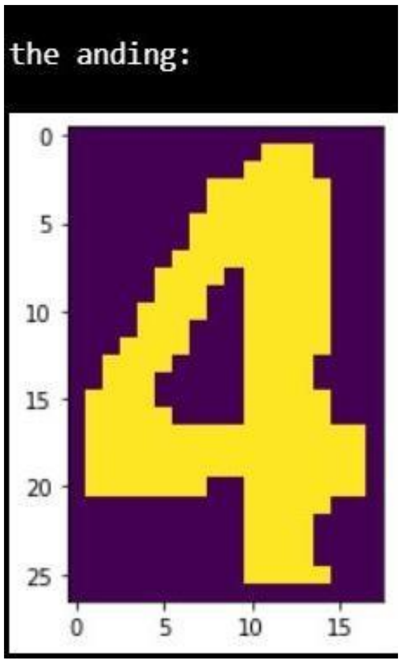
```
the template:
```

Figure 18 Template 2 image



```
the image:
```

Figure 17 Original Image



```
the anding:
```

Figure 16 ANDing between them(T2)

e. keep track of the highest ratio and the corresponding digit label by comparing the ratio of common pixels with the highest ratio seen so far (**“max_val”**). If the current ratio is higher than the best seen so far, the function updates **“max_val”** and **“best_digit”** to the current ratio and the corresponding digit label, respectively.

5- After comparing the image patch with all template images for all digits, check if the best match digit label is correct by comparing it with the ground truth label for the current bounding box. If the best match digit label is not None, the function compares it with the ground truth label for the current bounding box. If they match, the function increments the **“accuracy”** variable.

6- Draw a rectangle around the identified digit on the input image

7- Finally, return the value of **“accuracy”**, which represents the number of correctly identified digits in the input image.

Overall, this function uses a set of template images to identify digits in an input image using adaptive thresholding and bitwise operations. It then evaluates the accuracy of the identification by comparing the identified digit labels with the ground truth labels for a set of bounding boxes.

Algorithm’s Limitations

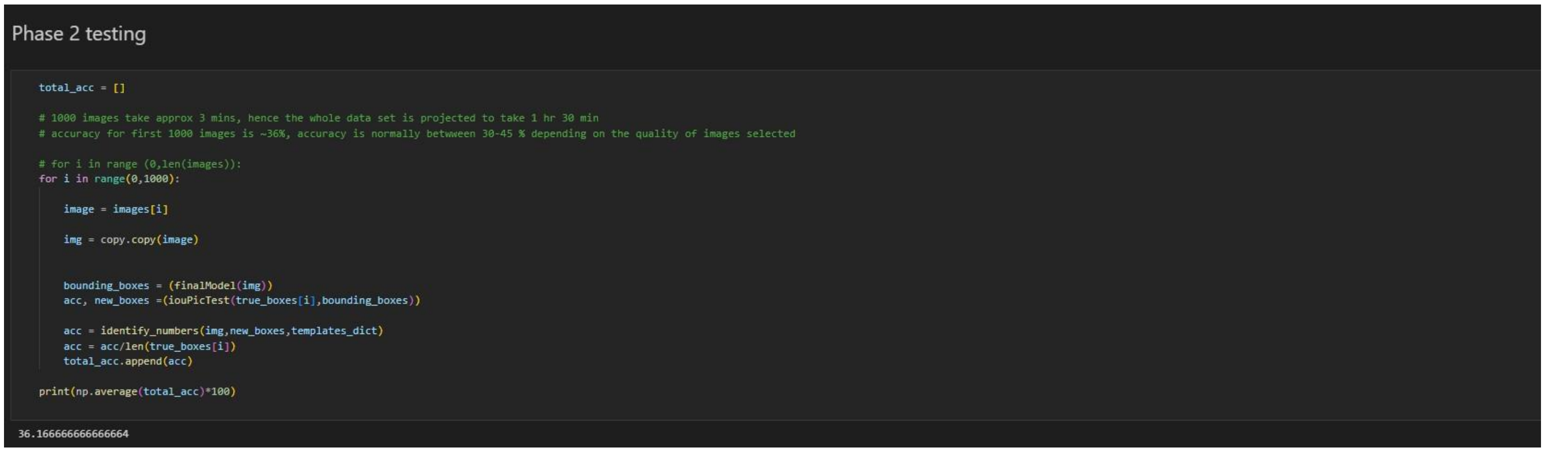
Algorithm might not be efficient in recognizing digits in an image in these cases:

1- When different digits in an image are very similar in shape or structure (ex: 6 and 8 , 6 and 9, 7 and 1)

2- Quality of image affects the result of algorithm in the pixels counted in Step 4-d

3- Extreme cases are not covered in the template used even though it has many variations of each digit , but any digit that has different font/size/shape that doesn’t exist in the template will not be recognized

Output Images and Accuracy for the Algorithm



Phase 2 testing

```
total_acc = []

# 1000 images take approx 3 mins, hence the whole data set is projected to take 1 hr 30 min
# accuracy for first 1000 images is ~36%, accuracy is normally betwween 30-45 % depending on the quality of images selected

# for i in range (0,len(images)):
for i in range(0,1000):

    image = images[i]

    img = copy.copy(image)


    bounding_boxes = (finalModel(img))
    acc, new_boxes =(iouPicTest(true_boxes[i],bounding_boxes))

    acc = identify_numbers(img,new_boxes,templates_dict)
    acc = acc/len(true_boxes[i])
    total_acc.append(acc)

print(np.average(total_acc)*100)
```

```
36.166666666666664
```

Figure 22 Accuracy of algorithm

The accuracy reached in Phase 2 is 36.164%

Sample 1:

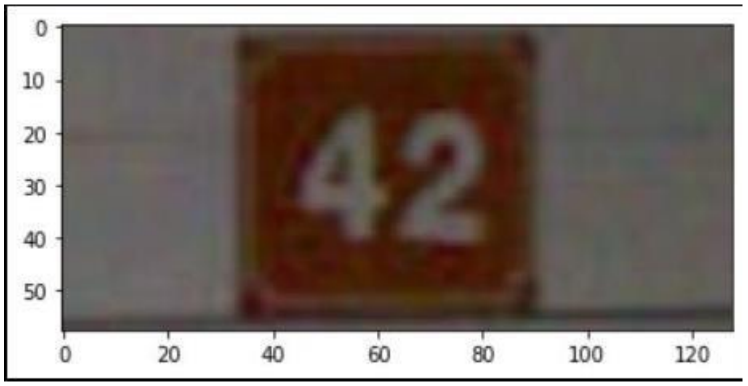


Figure 24 Original Input Image

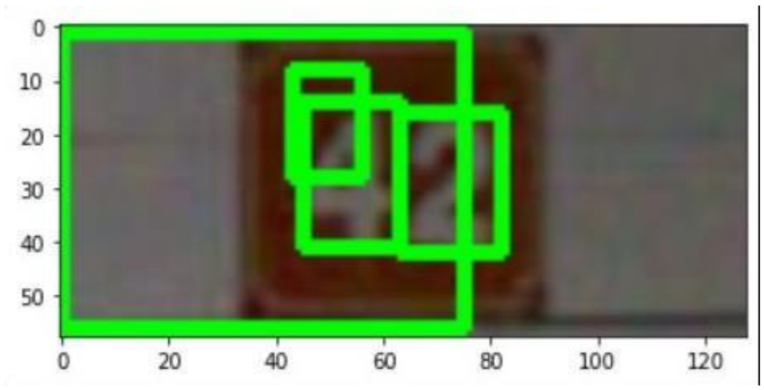


Figure 23 Phase 1 Output

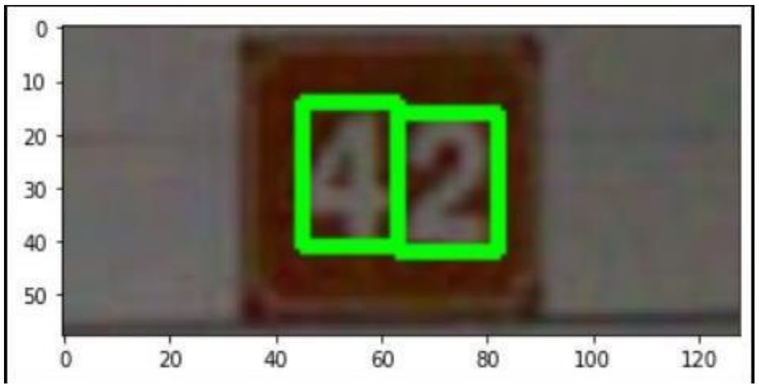


Figure 26 Phase 1 output with accurate boxes only

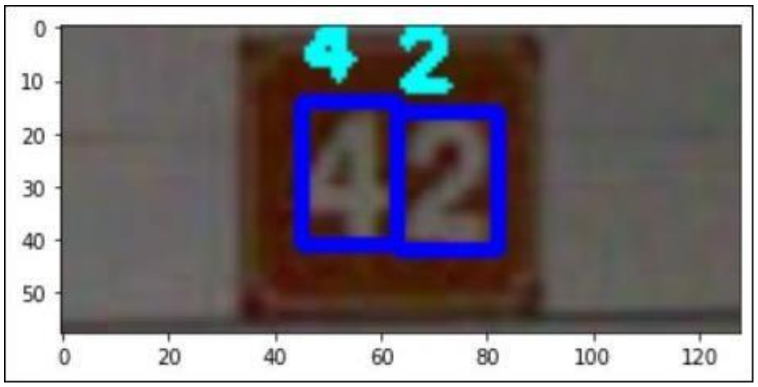


Figure 25 Recognized Digits

Sample 2

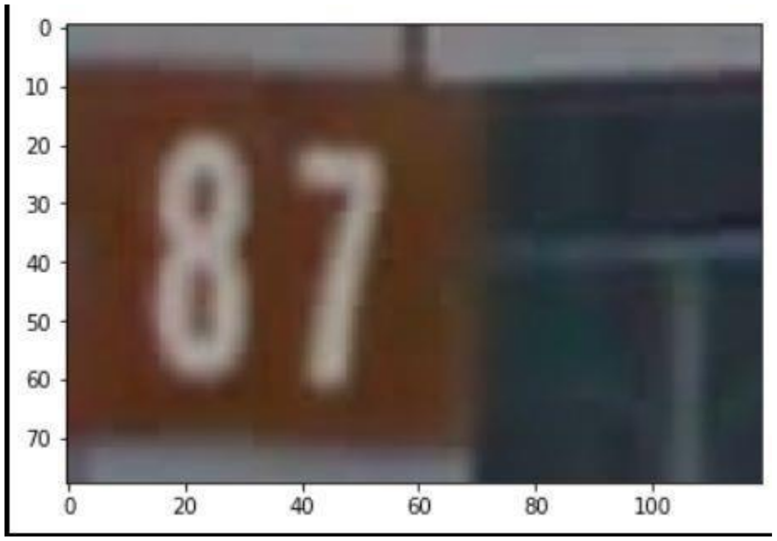


Figure 27 Original Image

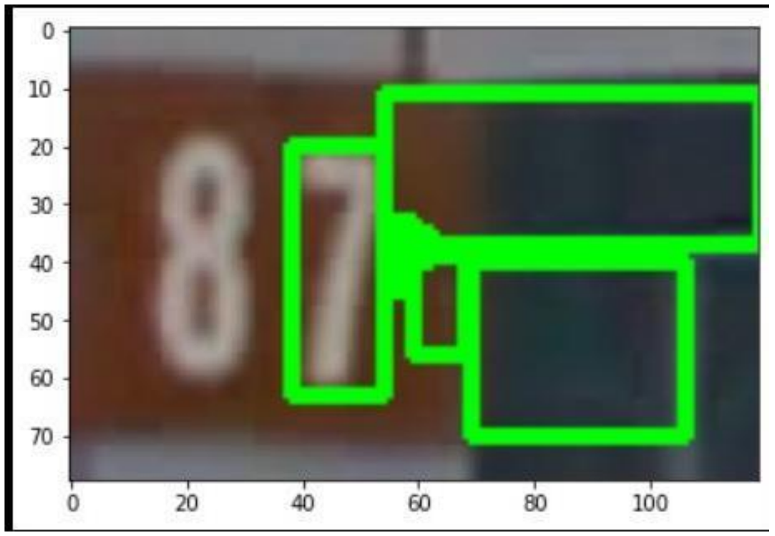


Figure 28 Phase 1 output

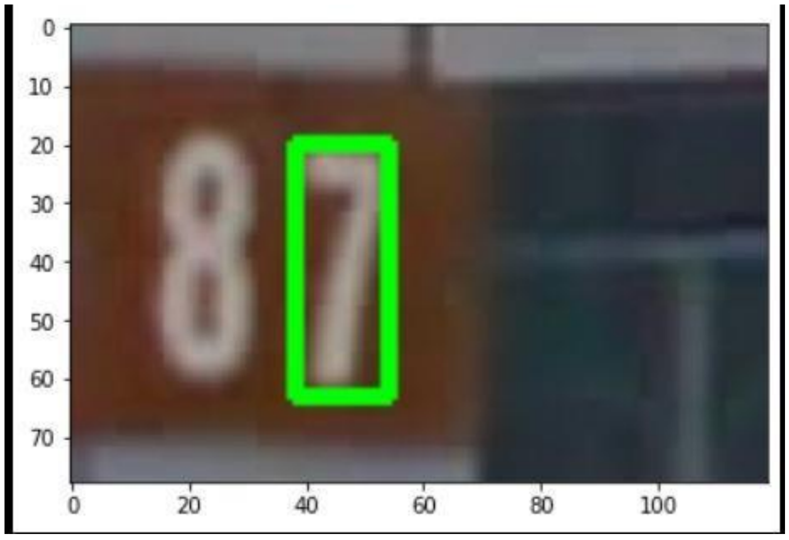


Figure 30 Phase 1 output with accurate boxes only

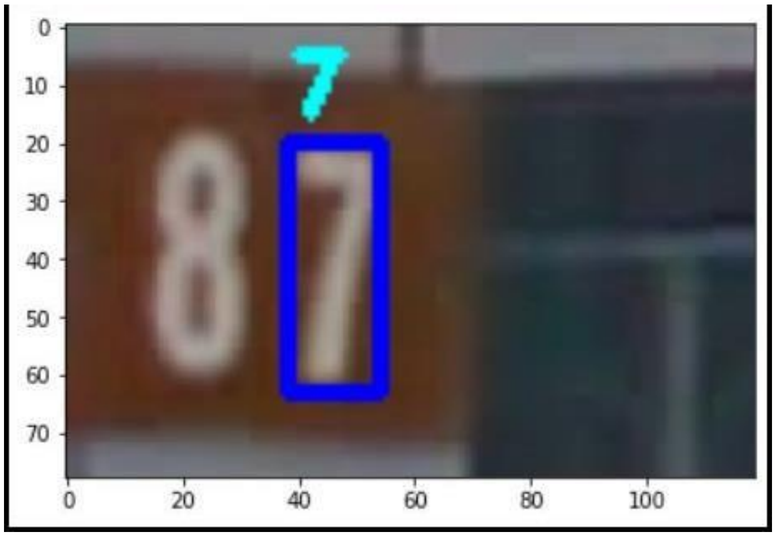


Figure 29 Recognized Digits

Sample 3 (Blurred Image limitation)

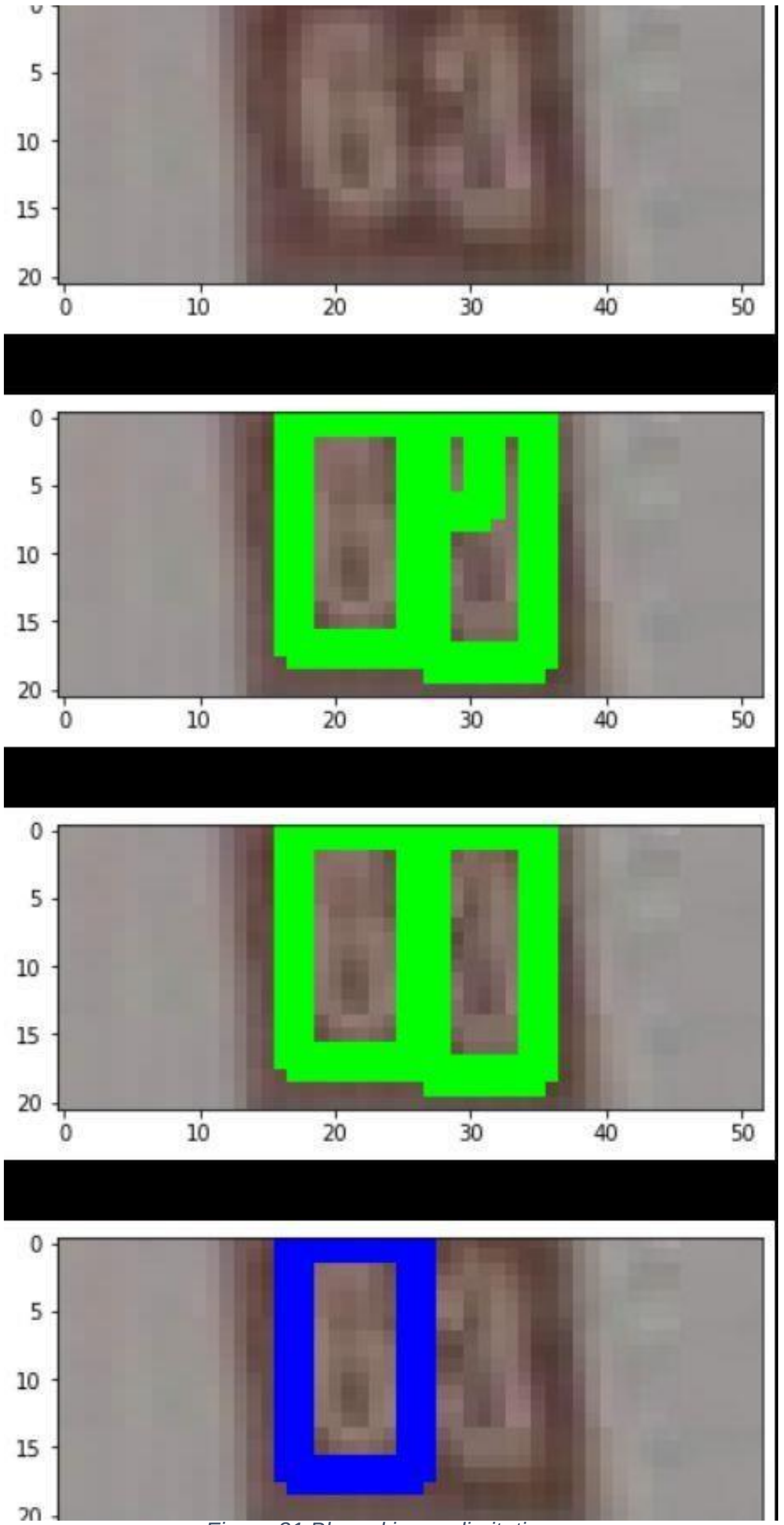


Figure 31 Blurred image limitation

Sample 4 : (Blurred image limitation)

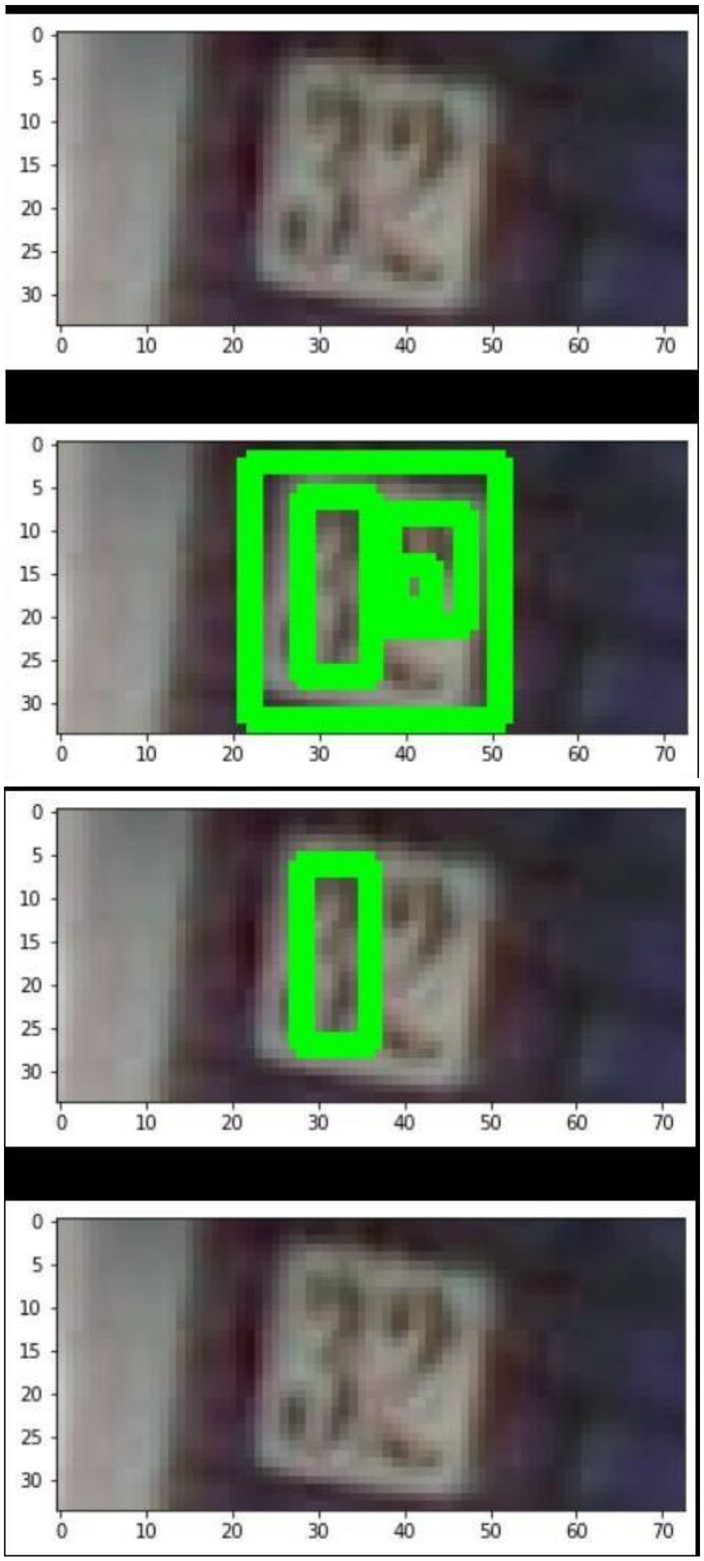


Figure 32 Blurred image limitation

Sample 5: (Digits with fonts not in the template images limitation)

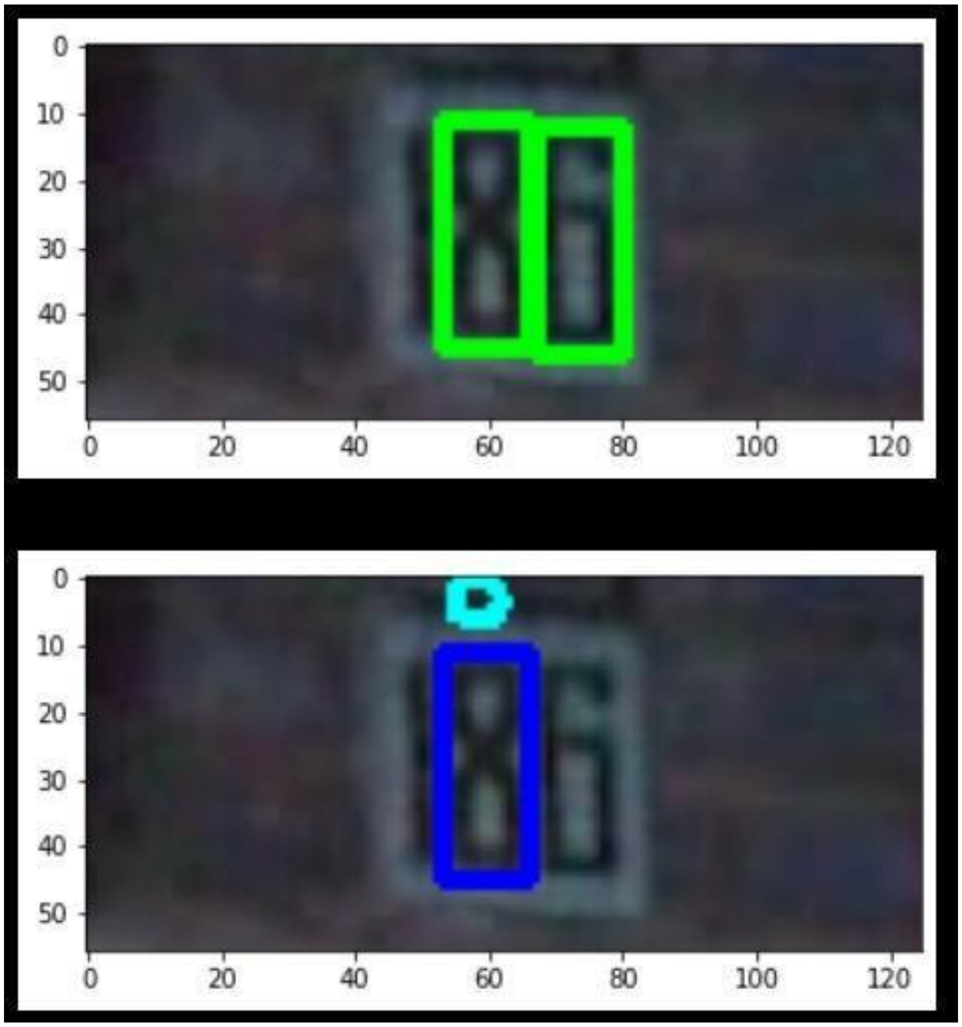
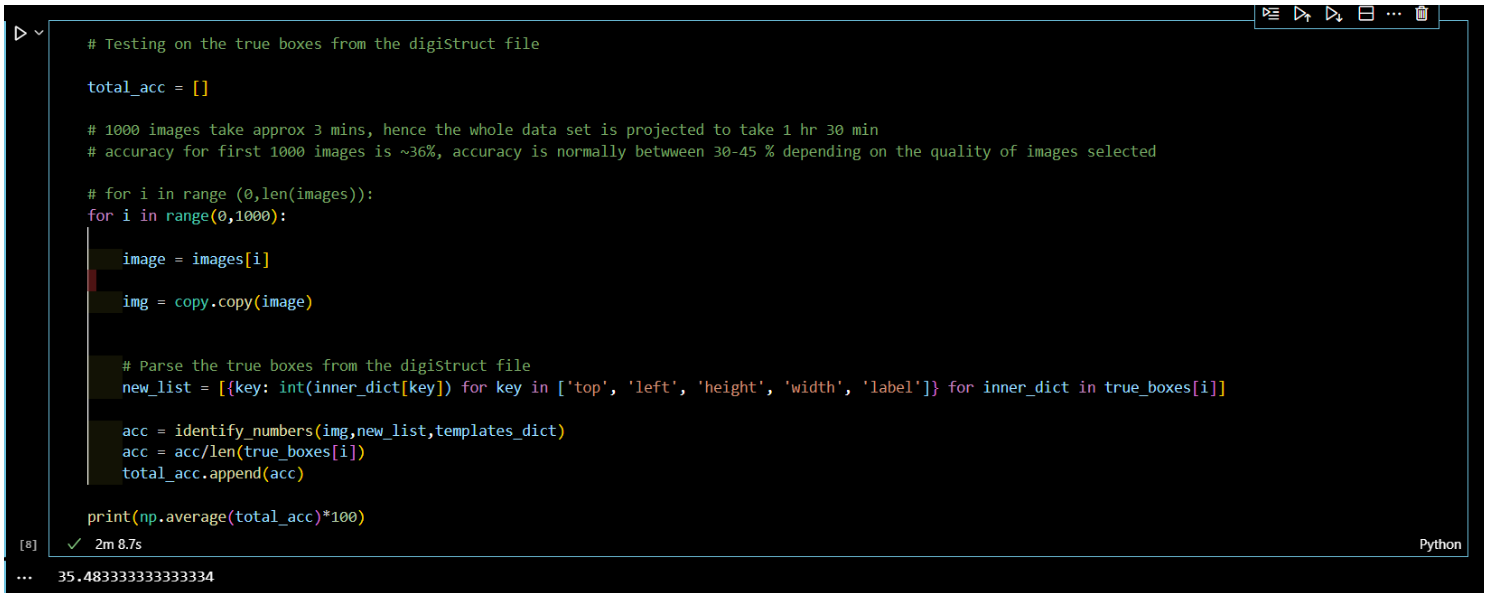


Figure 33 Digits of similar shape limitation

Phase 2 only

We tried phase 2 separately by using the true boxes that were in the digitStruct file.



```python
# Testing on the true boxes from the digiStruct file

total_acc = []

# 1000 images take approx 3 mins, hence the whole data set is projected to take 1 hr 30 min
# accuracy for first 1000 images is ~36%, accuracy is normally betwween 30-45 % depending on the quality of images selected

# for i in range (0,len(images)):
for i in range(0,1000):

    image = images[i]

    img = copy.copy(image)


    # Parse the true boxes from the digiStruct file
    new_list = [{key: int(inner_dict[key]) for key in ['top', 'left', 'height', 'width', 'label']} for inner_dict in true_boxes[i]]

    acc = identify_numbers(img,new_list,templates_dict)
    acc = acc/len(true_boxes[i])
    total_acc.append(acc)

print(np.average(total_acc)*100)
```

```
[8]  ✓ 2m 8.7s                                                Python
```

```
35.483333333333334
```

As we can see it resulted in 35% accuracy which is almost the same as to that of phase1 output into phase 2.

Conclusion

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

Github Repository

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