

# Code Explanation Report

SVHN-Recognition

By: Omar Muhammad

## Contents

Introduction .....	3
Purpose of the code .....	3
Phase 1 Localization .....	5
Code Overview .....	6
Dependencies Used .....	6
Hight Level Workflow .....	7
Code Explanation .....	8
Load Dataset .....	8
Estimate Digit Area.....	9
Canny Edge.....	11
Localize Digits.....	13
Get Intersection Percentage.....	15
Find Encompassing Rectangle .....	17
Clear Dir .....	19
Localize Dir .....	21
Running The Code: .....	23
Testing The Code: .....	24
Phase 2 Recognition .....	25
Code Overview .....	26
Dependencies Used .....	26
Hight Level Workflow .....	27
Code Explanation .....	28
Load Dataset .....	28
Normalize Image: .....	29
Extract Features: .....	31
Match Features: .....	33
Test Images:.....	35
Running The Code: .....	37
Testing The Code .....	38

## Introduction

In the contemporary urban landscape, the accurate and efficient recognition of house numbers stands as a crucial challenge with far-reaching implications. The precise identification of house numbers holds significance across diverse domains including navigation, emergency response, mail delivery, and location-based services. However, the traditional methods of manual address parsing and recognition have proven to be not only time-intensive but also prone to errors, underscoring the need for innovative solutions that can streamline the process while upholding a high degree of accuracy.

Amidst these challenges, the emergence of Street View House Number Recognition (SVHNR) technology offers a promising avenue for resolution. SVHNR harnesses the capabilities of computer vision and machine learning algorithms to automatically detect and decipher house numbers from images sourced from platforms like Google's Street View. By capitalizing on these technological advancements, SVHNR has the potential to redefine our approach to address identification, providing a rapid, precise, and scalable alternative that can substantially enhance multiple facets of contemporary urban life.

This paper embarks on an exploration of the existing quandary associated with house number recognition, delving into the foundational principles that underpin the Street View House Number Recognition paradigm. Moreover, the paper delves into how this technology can be wielded to surmount the challenges intrinsic to conventional address identification techniques. Through an examination of the technical framework, advantages, and potential applications of SVHNR, we endeavor to underscore the transformative potential it bears in revolutionizing address recognition within our dynamic urban milieus. The subsequent sections of the paper delve into the technical intricacies of Street View House Number Recognition code, offering insights into its operational mechanics and its role in mitigating the persistent hurdles linked to precise address recognition.

## Purpose of the code

The primary purpose of the Street View House Number Recognition (SVHNR) code is to automate and enhance the process of identifying and extracting house numbers from images captured by street-level mapping platforms like Google's Street View. This code aims to address the challenges associated with manual house number recognition, which can be time-consuming, error-prone, and limited in scalability. By leveraging computer vision techniques and machine learning algorithms, the SVHNR code offers a reliable and efficient solution to accurately detect, localize, and interpret house numbers within images.

The code's purpose can be further elaborated upon as follows:

1. **Automated Recognition:** The SVHNR code is designed to automatically identify and extract house numbers from images without requiring manual intervention. This automation reduces the need for human effort in reading and recording house numbers, leading to increased efficiency and accuracy.
2. **Enhanced Accuracy:** Through advanced computer vision algorithms, the code aims to achieve a higher level of accuracy in house number recognition compared to manual methods. This improved accuracy is vital for ensuring that critical services such as emergency response, navigation, and mail delivery are directed to the correct addresses.
3. **Efficiency and Speed:** Manual address identification can be time-consuming, especially in densely populated urban areas. The SVHNR code accelerates the process by swiftly analyzing images and extracting house numbers, enabling quicker decision-making and response times.
4. **Scalability:** As urban environments continue to expand, the demand for efficient address recognition grows. The SVHNR code is designed to scale effortlessly, making it well-suited for applications in large cities and regions where numerous addresses need to be processed.
5. **Versatility:** The code's utility extends beyond traditional address identification. It can be integrated into various systems, including navigation apps, GIS (Geographic Information Systems) databases, logistics and delivery services, and urban planning tools.
6. **Reduced Errors:** Human errors in reading or recording addresses can lead to significant complications. The SVHNR code minimizes such errors by relying on consistent algorithms, contributing to improved data quality.
7. **Innovation in Urban Services:** By facilitating accurate and automated address recognition, the SVHNR code supports innovations in urban services and planning. It enables businesses and governments to make informed decisions based on reliable location data.


In essence, the purpose of the Street View House Number Recognition code is to harness the power of computer vision and machine learning to revolutionize address recognition. By doing so, it streamlines processes, enhances accuracy, and contributes to the advancement of urban services in our dynamic and rapidly evolving urban landscapes.

# Phase 1

# Localization

## Code Overview

### Dependencies Used



```
1  import sys
2  import math
3  import cv2
4  import os
5  import json
6  import numpy as np
7  import copy
```

## Hight Level Workflow

1. Load the Dataset:
  - Load the dataset from the 'training.json' file using the ``load_dataset`` function.
  - The dataset likely contains information about images and their corresponding ground truth bounding boxes for digits.
2. Digit Localization and Intersection Percentage Calculation:
  - For each test image in the 'test-images' folder:
  - Load the image using OpenCV.
  - Apply noise reduction using the ``canny_edge`` function.
  - Perform digit localization using the ``localize_digits`` function on the processed image.
  - Extract the ground truth bounding box information for digits from the dataset based on the filename.
  - Calculate the intersection percentage between the localized digits and the ground truth using the ``get_intersection_percentage`` function.
  - Store the calculated intersection percentage in a list.
3. Display and Save Localized Images (Optional):
  - If desired, display intermediate steps and localized images using the ``showSteps`` parameter in the ``localize_dir`` function.
4. Calculate and Display Accuracy:
  - Calculate the accuracy by averaging all intersection percentages obtained in step 2.
  - Print the calculated accuracy as a percentage rounded to one decimal place.
  - Print the list of individual intersection percentages for each test image.
5. Clear Localized Images Directory (Optional):
  - Clear the "localized-images" directory using the ``clear_dir`` function to remove previously saved localized images.
6. End of Workflow:
  - The workflow concludes with the display of accuracy results and optional localized image display based on your preferences.

This high-level workflow encapsulates loading the dataset, performing digit localization using edge detection, calculating intersection percentages, optional display and saving of localized images, calculating and displaying accuracy results, and clearing the localized images directory if needed.

## Code Explanation

### Load Dataset

```
1 def load_dataset(file_path: str):  
2     # Open the file in read-only mode  
3     f = open(file_path, 'r')  
4     # Load the contents of the file as JSON data  
5     data = json.load(f)  
6     # Return the loaded data  
7     return data
```

This code defines a Python function called `load\_dataset` that is used to load data from a JSON file. Let's break down the code step by step:

1. Function Definition:

i. `def load_dataset(file_path: str):`

- This line defines a function named `load\_dataset` that takes a single argument `file\_path` of type `str` (string). This argument is meant to be the path to the JSON file that you want to load.

2. Opening the File:

i. `f = open(file_path, 'r')`

- This line opens the file specified by the `file\_path` in read-only mode ('r'). It assigns the resulting file object to the variable `f`. This file object represents the opened file.

3. Loading JSON Data:

i. `data = json.load(f)`

- This line reads the contents of the opened file (stored in the `f` file object) and interprets it as JSON data. The `json.load()` function is used for this purpose. The resulting JSON data is stored in the variable `data`.



- Note: To make this code work, you need to import the `json` module at the beginning of your code using `import json`.

#### 4. Returning Loaded Data: return data

- After loading the JSON data, the function returns the loaded data by using the `return` statement. The data is typically a Python data structure (lists, dictionaries, etc.) that corresponds to the JSON content in the file.

In summary, this code defines a function `load\_dataset` that takes a file path as input, opens the specified file in read-only mode, loads its contents as JSON data, and returns the loaded data. To use this code, you would need to import the `json` module and make sure the file path you provide as an argument point to a valid JSON file.

## Estimate Digit Area

```
1 def estimate_digit_area(image_size):
2     # Estimate the maximum and minimum sizes of the digits based on the image size
3     # assume maximum digit height is 80% of the image height
4     max_digit_height = int(image_size[0] * 0.8)
5     aspect_ratio = [0.38, 0.51, 0.54, 0.53, 0.55, 0.58,
6                     0.53, 0.47, 0.57, 0.52] # aspect ratio of digits 0-9
7     # Assume maximum digit width is 90% of the image width, adjusted by the maximum aspect ratio
8     max_digit_width = int(image_size[1] * 0.9 * max(aspect_ratio))
9     # Assume minimum digit height is 10% of the image height
10    min_digit_height = int(image_size[0] * 0.1)
11    # Assume minimum digit width is 10% of the image width, adjusted by the minimum aspect ratio
12    min_digit_width = int(image_size[1] * 0.1 * min(aspect_ratio))
13
14    # Calculate the approximate maximum and minimum area of the digit contours based on the estimated sizes
15    max_digit_area = (max_digit_height * max_digit_width)
16    min_digit_area = (min_digit_height * min_digit_width)
17
18    return min_digit_area, max_digit_area
```

This code defines a function named `estimate\_digit\_area` that calculates the estimated minimum and maximum areas of digits within an image. The function takes `image\_size` as an argument, which is assumed to be a tuple representing the height and width of the image.

Let's break down the code step by step:

#### 1. Function Definition:

- i. `def estimate_digit_area(image_size):`
  - This line defines a function named `estimate\_digit\_area` that takes a single argument `image\_size`.

## 2. Maximum and Minimum Digit Sizes:

- i. `max_digit_height = int(image_size[0] * 0.8)`
  - This line estimates the maximum digit height as 80% of the image height.
- i. `aspect_ratio = [0.38, 0.51, 0.54, 0.53, 0.55, 0.58, 0.53, 0.47, 0.57, 0.52]`
  - This line defines a list `aspect\_ratio` that contains the aspect ratios of digits 0 to 9. The aspect ratio is the ratio of width to height for each digit.
- i. `max_digit_width = int(image_size[1] * 0.9 * max(aspect_ratio))`
  - This line estimates the maximum digit width as 90% of the image width, adjusted by the maximum aspect ratio.
- i. `min_digit_height = int(image_size[0] * 0.1)`
  - This line estimates the minimum digit height as 10% of the image height.
- i. `min_digit_width = int(image_size[1] * 0.1 * min(aspect_ratio))`
  - This line estimates the minimum digit width as 10% of the image width, adjusted by the minimum aspect ratio.

## 3. Calculating Digit Area:

- i. `max_digit_area = (max_digit_height * max_digit_width)`
  - This line calculates the approximate maximum area of a digit's contour (the area it occupies in the image) based on the estimated maximum digit height and width.
- i. `min_digit_area = (min_digit_height * min_digit_width)`
  - This line calculates the approximate minimum area of a digit's contour based on the estimated minimum digit height and width.

## 4. Returning Results:

- i. `return min_digit_area, max_digit_area`
  - This line returns a tuple containing the calculated minimum and maximum digit areas.

In summary, the `estimate\_digit\_area` function takes an image size as input, estimates the minimum and maximum areas that digits within the image might occupy, and returns these estimated values as a tuple. The code makes assumptions about the size and aspect ratios of digits to perform these calculations.

## Canny Edge

```
1 def canny_edge(img, showSteps=False):
2     # Reduce noise using bilateral filter
3     dst = cv2.bilateralFilter(img, 9, 75, 75)
4
5     # Display intermediate step if showSteps is True
6     if showSteps:
7         cv2.imshow('Noise Reduction', dst)
8         cv2.waitKey(0)
9         cv2.destroyAllWindows()
10
11     # Convert the image to grayscale
12     gray = cv2.cvtColor(dst, cv2.COLOR_BGR2GRAY)
13
14     # Display intermediate step if showSteps is True
15     if showSteps:
16         cv2.imshow('GrayScale', gray)
17         cv2.waitKey(0)
18         cv2.destroyAllWindows()
19
20     # Apply Canny edge detection algorithm
21     thresh = cv2.Canny(gray, 50, 100)
22
23     # Display intermediate step if showSteps is True
24     if showSteps:
25         cv2.imshow('Canny', thresh)
26         cv2.waitKey(0)
27         cv2.destroyAllWindows()
28
29     # Return the output image
30     return thresh
```

This code defines a function called `canny_edge` that performs Canny edge detection on an input image. The function takes an image (`img`) as its primary argument and has an optional argument `showSteps` that, if set to `True`, displays intermediate steps of the edge detection process using OpenCV's GUI window.

Let's break down the code step by step:

1. Function Definition:

i. `def canny_edge(img, showSteps=False):`

- This line defines a function named `canny_edge` that takes two arguments: `img`, which represents the input image, and `showSteps`, which is a boolean flag indicating whether to display intermediate steps during the edge detection process (default is `False`).

2. Bilateral Filtering for Noise Reduction:

i. `dst = cv2.bilateralFilter(img, 9, 75, 75)`

- This line applies a bilateral filter to the input image (`img`) to reduce noise. The `cv2.bilateralFilter()` function takes the image, a diameter, a value for color sigma, and a value for space sigma as its arguments.

3. Displaying Intermediate Steps (Optional):

i. `if showSteps:`

ii. `cv2.imshow('Noise Reduction', dst)`

iii. `cv2.waitKey(0)`

iv. `cv2.destroyAllWindows()`

- If `showSteps` is `True`, this block of code displays the intermediate image resulting from the bilateral filter operation using OpenCV's GUI windows.

4. Converting Image to Grayscale:

i. `gray = cv2.cvtColor(dst, cv2.COLOR_BGR2GRAY)`

- This line converts the filtered image (`dst`) to grayscale using the `cv2.cvtColor()` function.

5. Displaying Intermediate Steps (Optional):

i. `if showSteps:`

ii. `cv2.imshow('GrayScale', gray)`

iii. `cv2.waitKey(0)`

iv. `cv2.destroyAllWindows()`

- Similar to the previous step, if `showSteps` is `True`, this block of code displays the intermediate grayscale image.

6. Applying Canny Edge Detection: `thresh = cv2.Canny(gray, 50, 100)`
  - This line applies the Canny edge detection algorithm to the grayscale image (``gray``) using the ``cv2.Canny()`` function. The parameters ``50`` and ``100`` are the thresholds used for edge detection.
7. Displaying Intermediate Steps (Optional):
  - i. `if showSteps:`
  - ii. `cv2.imshow('Canny', thresh)`
  - iii. `cv2.waitKey(0)`
  - iv. `cv2.destroyAllWindows()`
  - Again, if ``showSteps`` is ``True``, this block of code displays the intermediate result of the Canny edge detection.
8. Returning the Output Image: `return thresh`
  - Finally, the function returns the edge-detected image.

In summary, the ``canny_edge`` function applies the Canny edge detection algorithm to an input image, optionally displays intermediate steps if requested, and returns the edge-detected image. The bilateral filter is used at the beginning to reduce noise in the image before applying edge detection.

## Localize Digits

```
1 def localize_digits(img):
2     # Find contours of the input image with external retrieval mode.
3     contours, _ = cv2.findContours(
4         img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
5
6     # Estimate the minimum and maximum area of digit contours based on the input image size using a separate function.
7     minArea, maxArea = estimate_digit_area(img.shape)
8
9     # Create an empty list to store final contours.
10    finalContours = []
11
12    # Iterate through all contours found earlier and keep only those that have an area between minArea and maxArea.
13    # Append the bounding rectangle of each selected contour to the finalContours list.
14    for contour in contours:
15        area = cv2.contourArea(contour)
16        if minArea < area < maxArea:
17            finalContours.append(cv2.boundingRect(contour))
18
19    # Return the list of bounding rectangles for localized digits.
20    return finalContours
```

This code defines a function named `localize_digits` that is used to localize (identify and extract) digit regions within a binary image, typically obtained through image processing techniques like edge detection. The function takes a binary image (`img`) as input, where white pixels represent the digit regions and black pixels represent the background.

Let's break down the code step by step:

1. Function Definition:

i. `def localize_digits(img):`

- This line defines a function named `localize_digits` that takes an image (`img`) as input.

2. Finding Contours:

i. `contours, _ = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)`

- This line finds contours within the input binary image using the `cv2.findContours()` function. The `cv2.RETR_EXTERNAL` retrieval mode retrieves only the external (outer) contours, and the `cv2.CHAIN_APPROX_SIMPLE` method compresses horizontal, vertical, and diagonal segments and leaves only their end points.

3. Estimating Minimum and Maximum Area: `minArea, maxArea = estimate_digit_area(img.shape)`

- This line estimates the minimum and maximum areas of digit contours based on the size of the input image using the previously defined `estimate_digit_area` function.

4. Creating Empty List: `finalContours = []`

- This line creates an empty list called `finalContours` to store the bounding rectangles of the localized digit regions.

5. Iterating Through Contours and Filtering:

i. `for contour in contours:`

ii. `area = cv2.contourArea(contour)`

iii. `if minArea < area < maxArea:`

iv. `finalContours.append(cv2.boundingRect(contour))`

- This loop iterates through the contours found in the image. For each contour, it calculates the area using `cv2.contourArea()`. If the area of the contour falls between the estimated minimum and maximum digit areas (`minArea` and `maxArea`), it means the contour likely corresponds to a digit. In that case, the bounding rectangle of the contour is obtained using `cv2.boundingRect()` and added to the `finalContours` list.

6. Returning Bounding Rectangles:

i. `return finalContours`

- After iterating through all the contours and selecting the ones with appropriate areas, the function returns the list of bounding rectangles corresponding to localized digit regions.

In summary, the `localize_digits` function takes a binary image, finds contours, filters the contours based on their areas using the estimated minimum and maximum areas, and returns a list of bounding rectangles that represent the localized digit regions within the image.

## Get Intersection Percentage

```
1 def get_intersection_percentage(myOutput, realOutput):
2     global HarshAccuracy
3
4     # Read in black.png as a grayscale image and create two copies of it.
5     img1Temp = cv2.imread('black.png', cv2.IMREAD_GRAYSCALE)
6     img1 = copy.deepcopy(img1Temp)
7     img2 = cv2.imread('black.png', cv2.IMREAD_GRAYSCALE)
8
9     # Create an empty list to store the intersection percentages.
10    allPercents = []
11
12    # Draw rectangles on img1 at the locations specified in realOutput.
13    for (x, y, w, h) in realOutput:
14        cv2.rectangle(img1, (x, y), (x + w, y + h), 255, 2)
15
16    # Draw rectangles on img2 at the locations specified in myOutput.
17    for (x, y, w, h) in myOutput:
18        cv2.rectangle(img2, (x, y), (x + w, y + h), 255, 3)
19
20    # Use a bitwise AND operation to calculate the intersection of img1 and img2.
21    interSection = cv2.bitwise_and(img1, img2)
22
23    # Calculate the IoU percentage and append it to allPercents.
24    if HarshAccuracy:
25        allPercents.append((np.sum(interSection == 255) /
26                             (np.sum(img1 == 255) + np.sum(img2 == 255) - np.sum(interSection == 255))) * 100)
27    else:
28        allPercents.append((np.sum(interSection == 255) /
29                             (np.sum(img1 == 255))) * 100)
30
31    # Shift the rectangles in realOutput by 5 pixels in each direction and calculate the IoU percentage again.
32    for shift in [[5, 0], [-5, 0], [0, 5], [0, -5]]:
33        img1 = copy.deepcopy(img1Temp)
34        for (x, y, w, h) in realOutput:
35            cv2.rectangle(
36                img1, (x + shift[0], y + shift[1]), (x + w + shift[0], y + h + shift[1]), 255, 2)
37            interSection = cv2.bitwise_and(img1, img2)
38            if HarshAccuracy:
39                allPercents.append((np.sum(interSection == 255) /
40                                     (np.sum(img1 == 255) + np.sum(img2 == 255) - np.sum(interSection == 255))) * 100)
41            else:
42                allPercents.append((np.sum(interSection == 255) /
43                                     (np.sum(img1 == 255))) * 100)
44
45    # Return the maximum IoU percentage.
46    return max(allPercents)
```

This code defines a function named `get_intersection_percentage` that calculates the Intersection over Union (IoU) percentage between two sets of bounding rectangles. This function compares two sets of rectangles: `myOutput` and `realOutput`. The `myOutput` rectangles are typically generated by some algorithm or process, while the `realOutput` rectangles are ground truth annotations representing the actual locations of objects.

Let's break down the code step by step:

1. Function Definition and Global Variable:
  - i. `def get_intersection_percentage(myOutput, realOutput):`
  - ii. `global HarshAccuracy`

- This line defines the function and introduces the global variable HarshAccuracy. The purpose of this variable becomes evident later in the code

## 2. Preparing Images and Lists:

- `img1Temp = cv2.imread('black.png', cv2.IMREAD_GRAYSCALE)`
  - `img1 = copy.deepcopy(img1Temp)`
  - `img2 = cv2.imread('black.png', cv2.IMREAD_GRAYSCALE)`
  - `allPercents = []`
- This block of code loads a grayscale image named 'black.png' and creates copies named `img1` and `img2`. It also initializes an empty list called `allPercents` to store the calculated intersection percentages.

## 3. Drawing Rectangles on Images:

- `for (x, y, w, h) in realOutput:`
  - `cv2.rectangle(img1, (x, y), (x + w, y + h), 255, 2)`
  - `for (x, y, w, h) in myOutput:`
  - `cv2.rectangle(img2, (x, y), (x + w, y + h), 255, 3)`
- This section draws rectangles on `img1` and `img2` at the locations specified by `realOutput` and `myOutput` respectively. These rectangles represent the bounding boxes of the objects.

## 4. Calculating Intersection over Union:

- `interSection = cv2.bitwise_and(img1, img2)`
  - 
  - `if HarshAccuracy:`
  - `allPercents.append((np.sum(interSection == 255) /`
  - `(np.sum(img1 == 255) + np.sum(img2 == 255) -`
  - `np.sum(interSection == 255))) * 100)`
  - `else:`
  - `allPercents.append((np.sum(interSection == 255) /`
  - `(np.sum(img1 == 255))) * 100)`
- This section calculates the IoU percentage between the two sets of rectangles. The code uses the `cv2.bitwise_and()` function to calculate the intersection of the two images (`img1` and `img2`). Depending on the value of the `HarshAccuracy` global variable, it calculates the IoU using slightly different formulas. The IoU represents the proportion of overlap between the rectangles.

## 5. Shifting Rectangles and Repeating

- `for shift in [[5, 0], [-5, 0], [0, 5], [0, -5]]:`
- `img1 = copy.deepcopy(img1Temp)`
- `for (x, y, w, h) in realOutput:`
- `cv2.rectangle(`
- `img1, (x + shift[0], y + shift[1]), (x + w + shift[0], y + h + shift[1]), 255, 2)`
- `interSection = cv2.bitwise_and(img1, img2)`



- vii.     if HarshAccuracy:
- viii.     allPercents.append((np.sum(interSection == 255) /
- ix.         (np.sum(img1 == 255) + np.sum(img2 == 255) -
- np.sum(interSection == 255))) \* 100)
- x.        else:
- xi.        allPercents.append((np.sum(interSection == 255) /
- xii.        (np.sum(img1 == 255))) \* 100)
- This loop performs the same IoU calculation as before, but with a shift applied to the rectangles in realOutput. It calculates the IoU with the original and shifted rectangles to account for small localization errors.

#### 6. Returning the Maximum IoU Percentage:

- i.     return max(allPercents)
- After calculating IoU percentages for different scenarios, the function returns the maximum IoU percentage from the allPercents list. This indicates the highest degree of overlap or accuracy between the myOutput and realOutput bounding boxes.

In summary, the `get_intersection_percentage` function calculates the IoU between two sets of bounding rectangles, accounting for possible small shifts in the ground truth bounding boxes. The degree of accuracy calculation can be adjusted using the `HarshAccuracy` global variable.

## Find Encompassing Rectangle

```

1  def find_encompassing_rect(rect_list):
2
3      # Initialize the minimum and maximum x and y values as infinite and negative infinite, respectively
4      min_x = float('inf')
5      min_y = float('inf')
6      max_x = -float('inf')
7      max_y = -float('inf')
8      # Initialize the maximum width and height values as negative infinite
9      max_w = -float('inf')
10     max_h = -float('inf')
11
12     # Iterate through each rectangle in the provided list of rectangles
13     for rect in rect_list:
14         # Extract the x, y, width, and height values from the current rectangle
15         (x, y, w, h) = rect
16         # Update the minimum x and y values if the current x or y value is smaller than the current minimum
17         min_x = min(min_x, x)
18         min_y = min(min_y, y)
19         # Update maximum x and y values
20         max_x = max(max_x, x + w)
21         max_y = max(max_y, y + h)
22         # Update maximum width and height values
23         max_w = max(max_w, w)
24         max_h = max(max_h, h)
25
26     # Calculate and return the minimum x, minimum y, width, and height values of the encompassing rectangle
27     return min_x, min_y, max_x - min_x, max_y - min_y

```

This code defines a function named `find_encompassing_rect` that calculates the bounding rectangle that encompasses a list of input rectangles. The function takes a list of rectangles (`rect_list`) as input and returns the coordinates and dimensions of the encompassing rectangle.

Let's break down the code step by step:

1. Function Definition:

i. `def find_encompassing_rect(rect_list):`

- This line defines a function named `find_encompassing_rect` that takes a single argument `rect_list`, which is a list of rectangles represented as tuples.

2. Initialization:

i. `min_x = float('inf')`  
ii. `min_y = float('inf')`  
iii. `max_x = -float('inf')`  
iv. `max_y = -float('inf')`  
v. `max_w = -float('inf')`  
vi. `max_h = -float('inf')`

- These lines initialize variables that will be used to keep track of the minimum and maximum coordinates (`x` and `y`) and dimensions (width and height) of the encompassing rectangle. The `float('inf')` value is used to ensure that the initial values can be properly updated in the loop.

3. Iterating Through Rectangles:

i. `for rect in rect_list:`  
ii. `(x, y, w, h) = rect`  
iii. `min_x = min(min_x, x)`  
iv. `min_y = min(min_y, y)`  
v. `max_x = max(max_x, x + w)`  
vi. `max_y = max(max_y, y + h)`  
vii. `max_w = max(max_w, w)`  
viii. `max_h = max(max_h, h)`

- This loop iterates through each rectangle in the input `rect_list`. For each rectangle, it extracts the coordinates (`x, y`) and dimensions (`w, h`). Then, it updates the minimum and maximum `x` and `y` values based on the current rectangle's coordinates, and updates the maximum width and height values based on the current rectangle's dimensions.

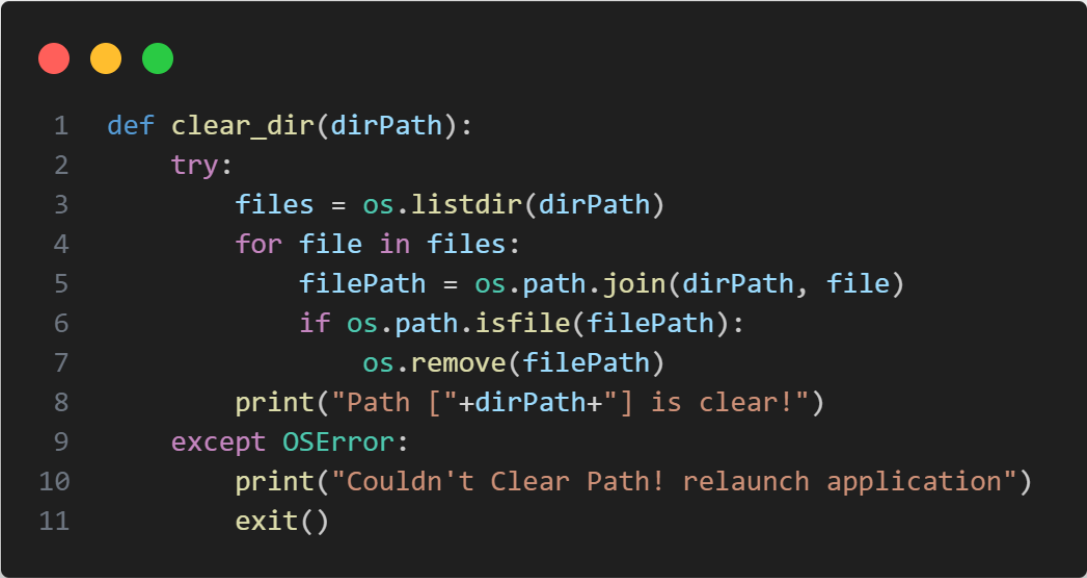
4. Calculating Encompassing Rectangle:

i. `return min_x, min_y, max_x - min_x, max_y - min_y`

- After iterating through all rectangles and collecting the necessary information, this line calculates and returns the coordinates (`x, y`) of the top-left corner of the encompassing rectangle and its width and height based on the differences between the maximum and minimum `x` and `y` values.

In summary, the `find_encompassing_rect` function takes a list of rectangles, finds the minimum and maximum coordinates and dimensions among them, and returns the coordinates and dimensions of the bounding rectangle that encompasses all the input rectangles.

## Clear Dir



```
1 def clear_dir(dirPath):
2     try:
3         files = os.listdir(dirPath)
4         for file in files:
5             filePath = os.path.join(dirPath, file)
6             if os.path.isfile(filePath):
7                 os.remove(filePath)
8             print("Path ["+dirPath+"] is clear!")
9     except OSError:
10        print("Couldn't Clear Path! relaunch application")
11        exit()
```

This code defines a function named `clear_dir` that is used to remove all files within a specified directory. The function takes a single argument `dirPath`, which is the path to the directory that you want to clear.

Here's a breakdown of the code:

1. Function Definition:

- i. `def clear_dir(dirPath):`

- This line defines a function named `clear_dir` that takes one argument: `dirPath`, which is the path of the directory to be cleared.

2. Removing Files from the Directory:

- i. `try:`

- ii. `files = os.listdir(dirPath)`

- iii. `for file in files:`

- iv. `filePath = os.path.join(dirPath, file)`

- v. `if os.path.isfile(filePath):`

- vi. `os.remove(filePath)`

- vii. `print("Path ["+dirPath+"] is clear!")`

- This section uses a try block to attempt the following operations:

- i. It uses `os.listdir(dirPath)` to obtain a list of filenames within the specified directory.
- ii. It iterates through each filename using a for loop.
- iii. It constructs the full path to the file using `os.path.join(dirPath, file)`.
- iv. It checks if the path corresponds to a file using `os.path.isfile(filePath)`.
- v. If the path does represent a file, it removes the file using `os.remove(filePath)`.

### 3. Handling Exceptions:

- i. `except OSError:`
  - ii. `print("Couldn't Clear Path! relaunch application")`
  - iii. `exit()`
- If an `OSError` exception occurs during the attempt to clear the directory, this section prints an error message and exits the application. This usually happens when the directory doesn't exist or there are permission issues.

In summary, the `clear_dir` function is responsible for clearing (deleting) all files within a specified directory. It iterates through the files in the directory, checks if each is a file, and deletes it. If any issue arises during this process, the function provides an error message and exits the application.

## Localize Dir

```
1 def localize_dir(dataset, showSteps):
2     percents = []
3     i = 0
4     total = len(os.listdir('test-images'))
5     clear_dir("localized-images")
6     for filename in os.listdir("test-images"):
7         # Display the loading progress.
8         i += 1
9         loadPercent = (i/total)*100
10        sys.stdout.write(f"\rLoading: [{ '=' * math.floor(loadPercent/10)}{' ' * (10 - math.floor(loadPercent/10))}] "
11                          f"{round(loadPercent, 1)}%")
12        f = os.path.join("test-images", filename)
13        imgReal = cv2.imread(f)
14
15        # Call the 'localize_digits' function to get the output.
16        myOutput = localize_digits(
17            canny_edge(imgReal, showSteps=showSteps))
18
19        # Extract the expected output from the 'dataset' parameter based on the filename.
20        realOutput = []
21        for box in dataset[int(filename.split(".")[0]) - 1]['boxes']:
22            realOutput.append((int(box['left']), int(
23                box['top']), int(box['width']), int(box['height'])))
24
25        # Calculate the intersection percentage and add it to the list.
26        percent = get_intersection_percentage(myOutput, realOutput)
27        percents.append(percent)
28
29        # If showSteps is True, display the localized image.
30        for (x, y, w, h) in myOutput:
31            cv2.rectangle(imgReal, (x, y), (x + w, y + h), (0, 255, 0), 2)
32            _ = cv2.imwrite("localized-images/"+filename, imgReal)
33        if showSteps:
34            cv2.imshow('Localized', imgReal)
35            cv2.waitKey(0)
36            cv2.destroyAllWindows()
37
38        # Return the list of intersection percentages.
39    return percents
```

This code defines a function named `localize_dir` that performs digit localization on a directory of images, evaluates the accuracy of localization, and optionally displays the process step-by-step. The function takes two arguments: `dataset` and `showSteps`.

Here's a breakdown of the code:

1. Function Definition:

- i. `def localize_dir(dataset, showSteps):`

- This line defines a function named `localize_dir` that takes two arguments: `dataset`, which is a dataset containing ground truth information about bounding boxes, and `showSteps`, a boolean indicating whether to display intermediate steps and localized images.

2. Initializing Variables and Clearing Directory:

- i. `percents = []`
  - ii. `i = 0`
  - iii. `total = len(os.listdir('test-images'))`

- iv. `clear_dir("localized-images")`
- These lines initialize variables for storing the calculated intersection percentages, counting the processed images, and getting the total number of images. The function `clear_dir` is used to clear the "localized-images" directory where the localized images will be saved.

### 3. Iterating Through Images:

- i. `for filename in os.listdir("test-images"):`
- ii. `# Display loading progress.`
- iii. `# ...`
- iv.
- v. `f = os.path.join("test-images", filename)`
- vi. `imgReal = cv2.imread(f)`
- This section iterates through each image in the "test-images" directory. It displays a loading progress bar indicating the percentage of images processed.

### 4. Localizing Digits:

- i. `myOutput = localize_digits(canny_edge(imgReal, showSteps=showSteps))`
- This line calls the `localize_digits` function to perform digit localization on the current image. It uses the `canny_edge` function to generate edge-detected images as input to the `localize_digits` function.

### 5. Extracting Ground Truth:

- i. `realOutput = []`
- ii. `for box in dataset[int(filename.split(".")[0]) - 1]['boxes']:`
- iii. `realOutput.append((int(box['left']), int(`
- iv. `box['top']), int(box['width']), int(box['height'])))`
- This section extracts the ground truth bounding boxes from the dataset based on the filename of the current image. It constructs a list of tuples representing (x, y, w, h) values of the boxes.

### 6. Calculating Intersection Percentage:

- i. `percent = get_intersection_percentage(myOutput, realOutput)`
- ii. `percents.append(percent)`
- This calculates the intersection percentage between the localized digits (`myOutput`) and the ground truth bounding boxes (`realOutput`). The calculated percentage is then added to the `percents` list.

### 7. Displaying Localized Images (Optional):

- i. `for (x, y, w, h) in myOutput:`
- ii. `cv2.rectangle(imgReal, (x, y), (x + w, y + h), (0, 255, 0), 2)`
- iii. `_ = cv2.imwrite("localized-images/"+filename, imgReal)`
- iv. `if showSteps:`
- v. `cv2.imshow('Localized', imgReal)`

- vi. `cv2.waitKey(0)`
- vii. `cv2.destroyAllWindows()`
- If `showSteps` is `True`, this section displays the localized image with bounding boxes drawn around the detected digits. It also saves the localized image in the "localized-images" directory.

#### 8. Returning Intersection Percentages:

- i. `return percents`
- After processing all images, the function returns the list of intersection percentages for each localized image.

In summary, the `localize_dir` function processes a directory of images for digit localization, compares the localized digits with ground truth data, calculates intersection percentages, and optionally displays intermediate steps and localized images. The result is a list of intersection percentages reflecting the accuracy of the localization process for each image.

### Running The Code:

```
1 # Load the dataset from the 'training.json' file using the 'load_dataset' function.
2 dataSet = load_dataset('training.json')
3 # Call the 'localize_dir' function to localize digits in the test images present in the 'test-images' folder.
4 percentages = localize_dir(dataSet, showSteps=False)
5 # Calculate accuracy by averaging all intersection percentages.
6 print(f"\n\nAccuracy is: {round(sum(percentages)/len(percentages), 1)}%")
7 print("Percentages:"+str(percentages))
```

You can run the localization phase by running the above code snippet that processes a dataset of images and calculates the accuracy of digit localization using a method of intersection percentages.

Here's the breakdown of each part:

#### 1. Loading the Dataset:

- i. `dataSet = load_dataset('training.json')`
- This line uses the `load_dataset` function to load the dataset from the 'training.json' file. The variable `dataSet` will contain the loaded dataset, likely in the form of a list of dictionaries, each dictionary representing information about an image.

#### 2. Localizing Digits and Calculating Percentages:

- i. `percentages = localize_dir(dataSet, showSteps=False)`
- This line calls the `localize_dir` function to perform digit localization on the test images. It takes the loaded dataset (`dataSet`) as input and sets `showSteps` to `False`, meaning that intermediate steps won't be displayed during the localization process. The `localize_dir`

function returns a list of intersection percentages, which is assigned to the variable percentages.

### 3. Calculating and Printing Accuracy:

- i. `print(f"\n\nAccuracy is: {round(sum(percentages)/len(percentages), 1)}%")`
  - ii. `print("Percentages:" + str(percentages))`
- This block of code calculates and prints the accuracy of the digit localization process. It computes the average intersection percentage by summing up all the percentages in the percentages list and dividing by the total number of percentages. The result is then rounded to one decimal place. Additionally, it prints the list of individual intersection percentages.

In summary, this code snippet loads a dataset of images, performs digit localization using the `localize_dir` function, calculates the accuracy based on intersection percentages, and prints the average accuracy along with the individual intersection percentages for each image.

## Testing The Code:

Here are snippets of some outputs of the localization code



Figure 1: Original Image

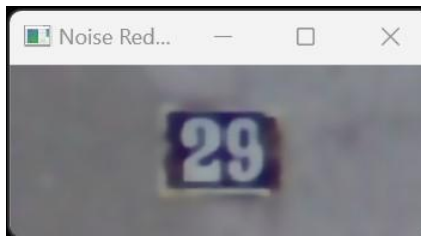


Figure 2: Noise Reduction Image

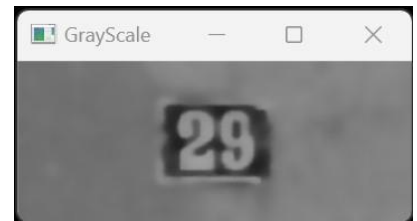


Figure 3: Gray Scale Image



Figure 4: Canny Edge Image

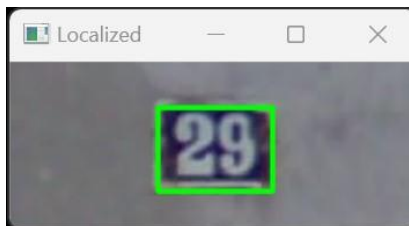


Figure 5: Localized Image

```
C:\Users\Omar\Documents\Python\house-plate-number-detector>
Path [localized-images] is clear!
Loading: [=====] 100.0%

Accuracy is: 34.0%
Percentages:[59.765625, 29.93311036789298, 40.1639344262295784, 67.0, 43.985849056603776, 53.493449781659386, 60.0]

C:\Users\Omar\Documents\Python\house-plate-number-detector>
```

Figure 6: Printouts




# Phase 2

# Recognition

## Code Overview

### Dependencies Used



```
1  import math
2  import cv2
3  import os
4  import sys
5  import numpy as np
6  import json
```

## Hight Level Workflow

1. Loading Dataset and Digit Templates:
  - Loads a dataset containing information about bounding boxes and labels for training images.
  - Loads and preprocesses digit template images used for matching.
2. Image Processing and Recognition Loop:
  - Iterates through the images in the 'test-images' directory to recognize digits within bounding boxes.
3. Normalization and Feature Extraction:
  - Normalizes the real images using the `normalize_image` function.
  - Extracts bounding box information from the dataset.
  - Extracts the region of interest (ROI) from the normalized image based on bounding box coordinates and dimensions.
4. Digit Template Comparison:
  - Compares the extracted ROI with digit templates using the `match_features` function.
  - Determines the best-matching digit template for each ROI.
5. Digit Prediction and Accuracy Calculation:
  - Predicts the digit within the ROI based on the best-matching template.
  - Compares the predicted digit with the label provided in the dataset.
  - Calculates accuracy by comparing predicted outcomes with actual labels.
6. Visualization (Optional):
  - If visualization is enabled (`v=True`), displays the original ROI image with the predicted digit labeled on it.
7. Total Accuracy Calculation and Return:
  - Calculates the overall accuracy by dividing the total number of correct predictions by the total number of predictions.
  - Returns the calculated accuracy as the result of the `test_images` function.

Overall, the functionalities of the code revolve around recognizing digits within bounding boxes of test images. It uses digit templates for matching, extracts features from images, calculates accuracy, and optionally provides visual feedback on the recognition process. The code simulates an OCR system by processing test images and evaluating its performance in digit recognition.

## Code Explanation

### Load Dataset

```
1 def load_dataset(file_path: str):  
2     # Open the file in read-only mode  
3     f = open(file_path, 'r')  
4     # Load the contents of the file as JSON data  
5     data = json.load(f)  
6     # Return the loaded data  
7     return list(data)
```

This code defines a Python function called `load\_dataset` that is used to load data from a JSON file. Let's break down the code step by step:

1. Function Definition:

i. `def load_dataset(file_path: str):`

- This line defines a function named `load\_dataset` that takes a single argument `file\_path` of type `str` (string). This argument is meant to be the path to the JSON file that you want to load.

2. Opening the File:

i. `f = open(file_path, 'r')`

- This line opens the file specified by the `file\_path` in read-only mode ('r'). It assigns the resulting file object to the variable `f`. This file object represents the opened file.

3. Loading JSON Data:

i. `data = json.load(f)`

- This line reads the contents of the opened file (stored in the `f` file object) and interprets it as JSON data. The `json.load()` function is used for this purpose. The resulting JSON data is stored in the variable `data`.

- Note: To make this code work, you need to import the `json` module at the beginning of your code using `import json`.

#### 4. Returning Loaded Data: return data

- After loading the JSON data, the function returns the loaded data by using the `return` statement. The data is typically a Python data structure (lists, dictionaries, etc.) that corresponds to the JSON content in the file.

In summary, this code defines a function `load\_dataset` that takes a file path as input, opens the specified file in read-only mode, loads its contents as JSON data, and returns the loaded data. To use this code, you would need to import the `json` module and make sure the file path you provide as an argument point to a valid JSON file.

#### Normalize Image:

```
1 def normalize_image(img):
2     # Convert the image to grayscale
3     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
4
5     # Split the grayscale image into separate planes
6     rgbPlanes = cv2.split(gray)
7
8     normalizedPlanes = []
9     for plane in rgbPlanes:
10        # Dilate the plane using a 7x7 kernel
11        dilatedImage = cv2.dilate(plane, np.ones((7, 7), np.uint8))
12
13        # Apply median blur with a kernel size of 21
14        blurredImage = cv2.medianBlur(dilatedImage, 21)
15
16        # Compute the absolute difference between the plane and the blurred image
17        planeDifferenceImage = 255 - cv2.absdiff(plane, blurredImage)
18
19        # Normalize the difference image to the range of 0-255
20        normalizedImage = cv2.normalize(planeDifferenceImage, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX,
21                                       dtype=cv2.CV_8UC1)
22
23        # Append the normalized plane to the list of normalized planes
24        normalizedPlanes.append(normalizedImage)
25
26    # Merge the normalized planes back into a single image
27    normalizedResult = cv2.merge(normalizedPlanes)
28
29    return normalizedResult
```

The provided code defines a Python function called `normalize_image` that takes an input image and applies a series of image processing steps to normalize the image's appearance. Let's break down the code step by step:

1. `gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)`
  - This line converts the input image `img` from its original color (presumably in BGR format) to grayscale using the `cv2.cvtColor()` function from the OpenCV library. The grayscale image is assigned to the variable `gray`.
2. `rgbPlanes = cv2.split(gray)`
  - This line splits the grayscale image `gray` into separate color planes (or channels). In this case, since the image is grayscale, it's actually splitting the image into multiple copies of the same plane. The resulting planes are stored in the list `rgbPlanes`.
3. The following steps are performed for each individual plane in `rgbPlanes`:
  - `dilatedImage = cv2.dilate(plane, np.ones((7, 7), np.uint8))`
    - i. Dilates the current plane using a 7x7 rectangular kernel. Dilating an image generally involves expanding the bright regions and is often used for noise reduction.
  - `blurredImage = cv2.medianBlur(dilatedImage, 21)`
    - i. Applies median blur to the dilated image using a kernel size of 21x21. Median blur is commonly used to reduce noise and smooth the image while preserving edges.
  - `planeDifferenceImage = 255 - cv2.absdiff(plane, blurredImage)`
    - i. Computes the absolute difference between the current plane and the blurred image. This difference image is then subtracted from 255, effectively inverting the intensities. This step may enhance certain features or patterns in the image.
  - `normalizedImage = cv2.normalize(planeDifferenceImage, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8UC1)`
    - i. Normalizes the difference image to the range of 0-255 using the `cv2.normalize()` function. This helps bring out details and contrast in the image.
  - The `normalizedImage` is appended to the `normalizedPlanes` list.
4. After processing all the planes, the code merges the normalized planes back into a single image using `normalizedResult = cv2.merge(normalizedPlanes)`.
5. Finally, the normalized result is returned as the output of the `normalize_image` function.

This function performs a series of image enhancement and normalization operations on the input image, potentially to highlight specific features or improve its visual quality. The effectiveness of these operations would depend on the characteristics of the input images and the goals of the processing.

## Extract Features:

```
1 def extract_features(img):
2     # Create a SIFT object for feature extraction
3     sift = cv2.SIFT_create()
4
5     try:
6         # Try converting the image to grayscale
7         gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
8     except:
9         # If conversion fails, assume the image is already grayscale
10        # Detect and compute key points and descriptors using SIFT
11        keyPoints, descriptors = sift.detectAndCompute(img, None)
12        return keyPoints, descriptors
13
14    # Detect and compute key points and descriptors using SIFT
15    # Return the detected key points and descriptors
16    keyPoints, descriptors = sift.detectAndCompute(gray, None)
17    return keyPoints, descriptors
```

The provided code defines a function called `extract_features` that uses the Scale-Invariant Feature Transform (SIFT) algorithm to detect and compute key points and descriptors for a given image. SIFT is a widely used algorithm for extracting distinctive features from images.

Let's go through the code step by step:

1. `sift = cv2.SIFT_create()`
  - This line creates a SIFT (Scale-Invariant Feature Transform) object for feature extraction. The `cv2.SIFT_create()` function initializes a SIFT instance that will be used for detecting key points and computing descriptors.
2. The code then enters a try block to handle possible exceptions when converting the image to grayscale:
  - a. `gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)`
    - i. This line attempts to convert the input image `img` from its original color format (BGR) to grayscale using the `cv2.cvtColor()` function. Grayscale images are often used for feature extraction as they simplify the processing.
  - b. If the conversion to grayscale fails (due to the image already being in grayscale or any other issues), the except block is executed:
    - i. `keyPoints, descriptors = sift.detectAndCompute(img, None)`

- I. The detectAndCompute() function of the SIFT object is called with the original color image img to detect key points and compute descriptors. Key points represent distinctive features in the image, and descriptors encode information about the neighborhood around each key point.
      - ii. The detected key points and descriptors are returned.
  3. If the grayscale conversion succeeds, the code continues to execute after the except block:
    - keyPoints, descriptors = sift.detectAndCompute(gray, None)
      - i. The detectAndCompute() function is called again, this time with the grayscale image gray as input. This function detects key points and computes descriptors for the grayscale image.
    - The detected key points and descriptors are returned.

In summary, the extract\_features function is designed to take an input image, detect distinctive features (key points) in the image, and compute descriptors that represent these features. It uses the SIFT algorithm for this purpose. If the image is already in grayscale, it directly detects key points and computes descriptors. If the image is in color, it first converts the image to grayscale and then proceeds with feature extraction.



## Match Features:

```
1 def match_features(img1, img2, v=False):
2     try:
3         # Create a brute-force matcher
4         brute_force_matcher = cv2.BFMatcher()
5
6         # Extract features (key points and descriptors) from both images
7         key_points_1, descriptors1 = extract_features(img1)
8         key_points_2, descriptors2 = extract_features(img2)
9
10        # Perform matching of descriptors between the two images
11        matches = brute_force_matcher.knnMatch(descriptors1, descriptors2, k=2)
12
13        # Perform ratio test to filter out ambiguous matches
14        optimizedMatches = [firstImageMatch for firstImageMatch, secondImageMatch in matches
15                             if firstImageMatch.distance < 0.75 * secondImageMatch.distance]
16
17        # Compute normalized scores and similarity sum
18        similarity_sum = 0.0
19        max_distance = float('-inf')
20        min_distance = float('inf')
21        for match in optimizedMatches:
22            distance = match.distance
23            similarity_sum += distance
24            if distance > max_distance:
25                max_distance = distance
26            if distance < min_distance:
27                min_distance = distance
28
29        # Compute the average normalized score as a measure of similarity
30        normalized_scores = [(max_distance - score) / (max_distance - min_distance + 0.000001) for score in
31                              (match.distance for match in optimizedMatches)]
32
33        # Draw the matched key points on the image (if enabled)
34        matched_image = cv2.drawMatches(img1, key_points_1, img2, key_points_2, optimizedMatches, None,
35                                         flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
36
37        if v:
38            # Display the matched image (if enabled)
39            cv2.imshow('Digit', matched_image)
40            cv2.waitKey(0)
41            cv2.destroyAllWindows()
42
43        return similarity_sum / len(normalized_scores) if normalized_scores else 0.0
44    except:
45        # Return infinity if an exception occurs during the process
46        return math.inf
```

The provided code defines a function called `match_features` that takes two input images (`img1` and `img2`) and performs feature matching using the Brute-Force Matcher with a ratio test. The purpose of this function is to measure the similarity between two images based on their key points and descriptors.

Let's break down the code step by step:

1. `try:`
  - The code begins with a try block to handle possible exceptions that might occur during the execution.
2. `brute_force_matcher = cv2.BFMatcher()`

- This line creates a Brute-Force Matcher object using the `cv2.BFMatcher()` constructor. The Brute-Force Matcher compares all descriptors from one image to all descriptors in the other image.
3. Extracting Features:
    - `key_points_1, descriptors1 = extract_features(img1)`
    - `key_points_2, descriptors2 = extract_features(img2)`
    - The code calls the `extract_features` function to extract key points and descriptors from both `img1` and `img2`.
  4. Matching Descriptors:
    - `matches = brute_force_matcher.knnMatch(descriptors1, descriptors2, k=2)`
    - The descriptors from both images are matched using the Brute-Force Matcher's `knnMatch` function with `k=2`, which returns the two best matches for each descriptor.
  5. Ratio Test:
    - The code filters the matches using a ratio test to keep only the matches that have a sufficiently better match compared to the second-best match. This is achieved by checking the ratio of the distances between the first and second matches.
  6. Similarity Calculation:
    - The code then calculates a similarity score based on the distances of the matched descriptors. It computes the normalized scores for each match based on their distances. The scores are normalized between 0 and 1, where a lower score indicates higher similarity.
  7. Visualization (if enabled):
    - The code optionally draws the matched key points on a new image using the `cv2.drawMatches()` function. This is controlled by the `v` parameter. If `v` is `True`, the matched image is displayed.
  8. Return Similarity:
    - `return similarity_sum / len(normalized_scores) if normalized_scores else 0.0`
    - The function calculates the average normalized score as a measure of similarity between the images. If there are no normalized scores (no matches or valid matches), it returns 0.0.
  9. Exception Handling:
    - If an exception occurs during the process (such as errors in feature extraction or matching), the function returns positive infinity (`math.inf`).

In summary, the `match_features` function performs feature matching between two images using the Brute-Force Matcher with a ratio test. It computes a similarity score based on the distances between matched descriptors and returns the average normalized score as a measure of image similarity.

## Test Images:

```
1 def test_images(v=False):
2     # Load the dataset file
3     print("Loading DataSet File..")
4     data_set = load_dataset('training.json')
5     print("DataSet File Loaded!!\n")
6     accuracy = []
7
8     # Preprocess digit templates
9     digit_templates = []
10    for digit_filename in os.listdir("digit-templates"):
11        template = os.path.join("digit-templates", digit_filename)
12        digit_template = cv2.imread(template)
13        digit_templates.append(digit_template)
14
15    directory = "test-images"
16    all_files = os.listdir(directory)
17    total = len(all_files)
18
19    for i, filename in enumerate(all_files, start=1):
20        load_percent = (i / total) * 100
21        if not v:
22            sys.stdout.write(
23                f"\rComputing Accuracy: [{i} * math.floor(load_percent / 10)]{' ' * (10 - math.floor(load_percent / 10))}"
24                f"{round(load_percent, 1)}%")
25
26        # Read the real image
27        image_real = cv2.imread(os.path.join(directory, filename))
28        # Get the bounding boxes for the current image
29        boxes = data_set[int(filename.split(".")[0]) - 1]['boxes']
30        # Normalize the real image
31        normalized_image = normalize_image(image_real)
32
33        for idx_box, box in enumerate(boxes):
34            # Extract the region of interest (ROI) from the normalized image
35            (x, y, w, h) = int(box['left']), int(
36                box['top']), int(box['width']), int(box['height'])
37            normalized_roi = normalized_image[y:y+h, x:x+w]
38
39            # Calculate similarities for all digit templates
40            similarities = []
41            for digit_template in digit_templates:
42                desired_height = h
43                aspect_ratio = digit_template.shape[1] / \
44                    digit_template.shape[0]
45                desired_width = int(desired_height * aspect_ratio)
46                resized_template = cv2.resize(
47                    digit_template, (desired_width, desired_height))
48                sim = match_features(resized_template, normalized_roi, v)
49                similarities.append(sim)
50
51            # Find the best match and get the corresponding digit
52            best_match_idx = similarities.index(min(similarities))
53            predicted_digit = os.listdir(
54                "digit-templates")[best_match_idx].split(".")[0]
55
56            if v:
57                # Display the predicted digit on the ROI image (if enabled)
58                image = cv2.cvtColor(
59                    image_real[y:y+h, x:x+w], cv2.COLOR_BGR2RGB)
60                aspect_ratio = image.shape[1] / image.shape[0]
61                image = cv2.resize(image, (int(500 * aspect_ratio), 500))
62                image = cv2.putText(image, predicted_digit, (image.shape[1] // 2, image.shape[0] // 2), cv2.FONT_HERSHEY_SIMPLEX,
63                    3, (0, 255, 0), 5, cv2.LINE_AA)
64                cv2.imshow('Digit', image)
65                cv2.waitKey(0)
66                cv2.destroyAllWindows()
67                print(f"\nImage {filename.split('.')[0]}, Box:{idx_box}: Label = {box['label']}, "
68                    f"Predicted Outcome = {predicted_digit}")
69
70            # Compute the accuracy by comparing the predicted digit with the label
71            accuracy.append(predicted_digit == str(box['label']).split(".")[0])
72
73    if not v:
74        sys.stdout.write(f"\rComputing Accuracy: [{i} * 10] 100%")
75    return sum(accuracy) / len(accuracy)
```

The provided code defines a function named `test_images` that processes a set of test images to recognize digits within bounding boxes. The function seems to be part of an Optical Character Recognition (OCR) system for digit recognition. Let's break down the code step by step:

1. `data_set = load_dataset('training.json')`
  - Loads a dataset from a JSON file named 'training.json'. The exact content of the dataset isn't shown here, but it seems to contain information about the bounding boxes and labels of digits.
2. Preprocessing Digit Templates:
  - The code reads digit template images from a directory named 'digit-templates' and stores them in the `digit_templates` list.
3. Loop through Test Images:
  - The code iterates through the files in the 'test-images' directory.
4. Loading Real Image:
  - Reads a real image from the 'test-images' directory.
5. Extracting Bounding Boxes:
  - Retrieves the bounding boxes for the current image from the loaded dataset.
7. Normalizing Real Image:
  - Normalizes the real image using the `normalize_image` function. The purpose of normalization is likely to enhance the features of the digits.
8. Loop through Bounding Boxes:
  - The code iterates through the bounding boxes within the current image.
9. Extracting Region of Interest (ROI):
  - Extracts the region of interest (ROI) from the normalized image based on the coordinates and dimensions of the bounding box.
10. Comparing with Digit Templates:
  - Compares the extracted ROI with digit templates using the `match_features` function to find the best match.

#### 11. Predicting Digit:

- Determines the best-matching digit template and predicts the digit within the ROI.

#### 12. Visualization (if enabled):

- If the `v` parameter is set to `True`, the code displays the original ROI image with the predicted digit labeled on it.

#### 13. Calculating Accuracy:

- Compares the predicted digit with the label provided in the dataset to compute accuracy.

#### 14. Returning Accuracy:

- Returns the calculated accuracy as the fraction of correct predictions.

The `test\_images` function essentially evaluates the accuracy of digit recognition on a set of test images with labeled bounding boxes. It uses digit templates and compares them to the normalized regions of interest in the test images, predicting the digit in each bounding box. The accuracy is calculated by comparing the predicted outcomes with the ground truth labels. If the `v` parameter is `True`, the function also provides visualization of the predicted outcomes on the test images.

### Running The Code:

```
1  acc = test_images(v=False)
2  print(f"\n\nAccuracy: {round(acc * 100, 1)}%")
```

You can run the recognizer by running this code snippet that evaluates the accuracy of the OCR (Optical Character Recognition) system implemented in the previous code. It uses the `test\_images` function to calculate the accuracy and then prints the accuracy as a percentage.

Here's a breakdown of the code:

1. ``acc = test_images(v=False)``:
  - This line calls the ``test_images`` function with the argument ``v=False``, which means that visualization of the recognition process will be turned off (no images will be displayed during processing).
  - The function returns the accuracy of the OCR system, which is assigned to the variable ``acc``.
2. ``print(f"\n\nAccuracy: {round(acc * 100, 1)}%")``:
  - This line prints the calculated accuracy in a human-readable format.
  - It uses an f-string to format the output string.
  - ``round(acc * 100, 1)`` calculates the accuracy value multiplied by 100 and rounds it to one decimal place. This converts the accuracy from a fraction to a percentage.
  - The formatted output displays the accuracy percentage along with the word "Accuracy".

In summary, this code evaluates the accuracy of the OCR system on a set of test images, and then prints the accuracy percentage to the console. The accuracy reflects how well the system performed in recognizing digits within the images. The ``v=False`` argument ensures that no visualizations are displayed during the accuracy calculation.

## Testing The Code

Here are snapshots of the output of running the recognizer code on some images

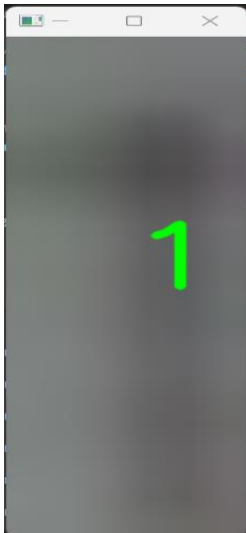


Figure 7: predict digit 1



Figure 8: predict digit 1

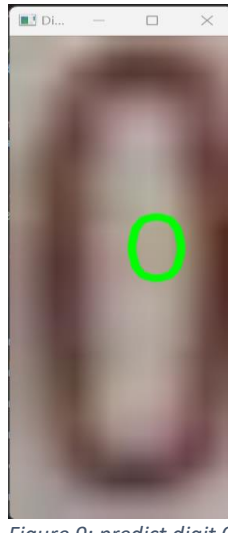


Figure 9: predict digit 0



Figure 10: predict digit 4

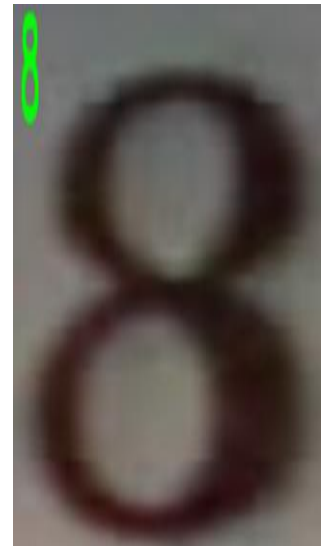


Figure 11: predict digit 8