Traffic Sign Detection and Classification

Diogo Gomes
up201806572@up.pt
Pedro Coelho
up201806802@up.pt
Nuno Santos
up201405774@up.pt

Faculty of Engineering
University of Porto
Porto, PT

Abstract

In the current day and age, the ability to recognize certain features in an image or video are being used more and more, with the development of a higher amount of intelligent AIs with the several purposes: self driving, autonomous sensor, explicit censorship in websites or other applications, identity recognition, etc. As those needs evolve we must understand what steps are needed to identify a pre-selected feature on an image, such as road signs on a dash cam footage, which is the theme of this project.

1 Introduction

For this project we are trying to find and classify road signs present in several images on a data set. These signs can be of several types, such as blue squares, red circles, stop signs, blue circles and triangles. However, throughout the project we only pursued the main objective which was the identification of the first three (blue squares, red circles and stop signs). In order to reach that end, different pipelines, with different methods applied to them were created and applied on a data set containing an array of images like was said previously.

2 Data Set

The data set used on this project consists of 88 images containing blue squares, 81 images containing red circles and 51 images of stop signs. Some of these images contain more than one traffic sign and are present in 2 or more of these categories. For the effect of an easier evaluation of the final results this data set was divided into three different directories (red_circles, blue_squares and stop_signs) where the images were organized by the types of signs contained in them.

2.1 Image conditions

The images present in the data set have good resolution, they are 300px by 400px on average and don't deviate to far from these dimensions. Our challenge in identifying traffic signs is made harder by differences in lighting, slanted sings, signs partially hidden and multiple signs present in the same image.

3 Assumptions accepted

Considering the basis of this project which is to identify what type of signs are present in a image in a data set. The signs we need to identify are of three types: red circles, blue squares/rectangles and stop signs. The base assumptions are that all signs are well lit, they're facing the camera on a 90º angle, they are not obstructed and there's only one sign to identify in each image. As we will show later on, we are classifying these cases with fairly good accuracy but we are also considering some deviations given that we are also able to classify some more complex situations where the base assumptions are not completely true.

4 Detection of Candidate Regions

To detect the candidate regions of the images we segment the colors red and blue separately. For each image an HSV version of it is created and is then filtered with a mask to detect the desired color and turn the image into grey scale, pixels are transformed to white when they match the desired color.

OpenCV uses an angle of 180º to define the hue of a color instead of the regular 360º. For the color red we have to use 2 masks because on the HSV color model red shows up at both ends of the hue spectrum. To detect it we accepted values from (0,90,50) to (10,255,255) for the lower part of the mask and accepted values from (160,90,50) to (180,255,255) for the upper part of the red mask.

For the color blue only 1 mask was needed any color between the values of (90,140,50) to (130,255,255) was accepted.

These values were decided based on our tests to detect the colors we desired. Our goal with this section was to catch as much of the traffic signs as possible even if sometimes the background also came attached to our grey scale image as a consequence. This is dealt with further down the pipeline.

5 Algorithm selection and application

To decide what algorithms or filters work best for our pretended goal we need to understand what features are we trying to identify. Considering our chosen solution, the objective of applying any filter to the image has the end goal of detecting the different signs with different shapes (circles,rectangles or octagons). All decisions were made after studying and considering results of several algorithms.

The final solution comprised on the usage of Bilateral Filtering which consists of a a smoothing algorithm that preserves edges well, Canny Edge Detection for edge detection (and better delineating of contours of the images and the signs in them), Morphological Transformations for closing effects on the signs with letters (advantageous mainly for circle signs so that letters like 'O' can't be detected as a circle) and for signs with missing parts (so that contours are accurate) and Hough Circles Transform for detecting circles, as it will be explained how they were applied in the pipeline section.

6 Pipeline

In this section, the pipelines used to detect each of the traffic sign types (red circles, stop signs and blue squares/rectangles) respectively, will be minutely described by delineating all the steps and the sole reason on why they were taken on the different cases. These steps consist on the appliance of filters (segmentation of colors), the use of edge detection algorithms, feature extraction methods and morphological transformations.

For every image a small amount of smoothing (2nd image of each annex image describing the pipeline) is applied to remove trace amounts of noise that may be present in the picture and could lead to failures in identifying the features.

After this the image segmentation is performed as described in section 3 Detection of Candidate Regions according to the type of traffic sign being searched.

After this is done the segmented image is processed in an attempt to identify the traffic sings. Another round of smoothing is done to remove any unnecessary color noise and help differentiate important features from the image.

6.1 Contours

For each traffic sign a slightly different approach is used and some methods are applied which will be described in each of the following sections. After that extra processing is done it is time to detect where exactly are the desired features in the image.

This is done by using a contour finder. Our contour finder is provided by the openCV module and it traces the edges of a grey scale image. This contour finder return multiple contours catching anything that has white pixels. This returns many uninteresting results and some filtering is performed. Contours which are too small to be any traffic sign in readable distance are ignored. Very irregular contours with too many edges are also discarded as they are most likely a contour of noise and not a clean traffic sign. After this we select the contour with the largest area that matches these criteria and this seems to be a good heuristic for our data set.

6.2 Edge counter

To differentiate the various contours found in our images we used a function provided by the openCV module approxPolyDP() which takes a contour and counts the number of straight edges. This proves especially useful in finding what type of sign is present in the image. Depending on the number of edges returned we can classify the traffic signs.

6.3 Red Circles

On the red circles pipeline firstly we used the segmentation method described in the Detection of Candidate Regions section and obtained a red segmented gray scale image (3rd image of each annex image describing the pipeline of a red circle or stop sign) with the reds isolated (in most cases). After isolating the reds we perform the contour detection. When identifying circles with our edge detector the circles tend to have a number of edges from 10 to 17 or 18. The number of edges rarely deviate from these values meaning this is a good metric to avoid false positives in the case of red circles. When counting the number of edges, if the program counts a number above 9 then the image is subjected to the circle finder. The circle finder performs an operation of closing, this means a dilation followed by an erosion to attempt to smooth the edges and remove possible details which would be harmful in detecting a clean circle.

After this the function HoughCircles provided by the openCV module is used to detect the circles and draw them. This function tries to fit the provided features of a grey scale image into circles with some level of tolerance for deformations or slanted perspectives. If it does recognize a possible red circle then the image with the drawn circle (final image of annex images describing the pipeline of red circles) is placed on the red circles folder found within the results directory.

6.4 Stop Signs

For the stop signs pipeline the approach was similar to the red circles pipeline, only the Hough circles transform was obviously not applied and the criterion for the classification of them was the amount of sides on the polygon approximate being equal to 8 as stop signs are octagonal (final image of figure 3 in annexes). The described steps aren't tolerable in most of the following situations:

- **occlusions** of vertices (new edges created; more than 8)
- **further away signs** since they are small in the image and may lead to the detection of more edges and become similar to circles

Therefore our model doesn't detect most stop signs in those conditions.

6.5 Blue Squares/Rectangles

The pipeline, used for detecting blue square/rectangle traffic signs, firstly uses the segmentation method (described in the Detection of Candidate Regions section) for the blue colour (3rd image of annex image describing the pipeline of a blue square), then applies a closing morphological transformation (5th image of figure 6 in annexes) to prevent the signs from having missing parts, crucial for the contouring and classification phase of the pipeline. After the appliance of the morphological transformation, Canny edge detection is applied (6th image of figure 6 in annexes) for smoother correlated edges, resulting on better contours and edge counting. Finally, the sign will be classified as a blue square when the number of edges is equal to 4 (final image of figure 6 in annexes).

Like stop signs the described steps are intolerable to occlusions of vertices (new edges created).

7 Results

To get the results 4 directories were made where the different signs were organized by their predicted classifications. Besides the three types of signs described before, the images could be classified as unrecognised when no signs were detected. So the directories created were the following:

- **red_circles** for the images classified by having at least one red circle
- **blue_squares** for the images containing at least one blue square / rectangle
- **stop_signs** for the images containing at least one stop sign
- **unknowns** for the images classified as unrecognised

To evaluate our different solutions and parameters used in different algorithms the number of unrecognized signs was obtained and an accuracy function calculated (by dividing the amount of correct classifications by the amount expected) was utilized.

Throughout the work done on the project two solutions were achieved. The first one runs the pipeline once for each sign color, red and blue, with the same amount of smoothing applied to the original image (bilateral filter applied once). The second one does the same as the first one, but when no signs are detected on a pipeline runs the same pipeline again only with more smoothing applied to the original image. This last solution did better than the first one because it does a finer job at distinguishing colors in cases where the background color is similar to the sign color. These are the metrics obtained for each solution:

Smoothing	Red Circles	Blue Squares	Stop Signs	Unrecognised
Normal	40.0%	47.1%	66.7%	62
High	52.0%	52.9%	68.6%	49

Table 1: Accuracy of the different solutions and amount of unrecognized by smoothing amount applied on original image

8 Degree of fulfillment of the task

Our major goal for this solution was to be able to identify and classify images with one sign, well lit and no obstructions (base conditions). In that regard our solution was able to meet the aims, in which all images that fit in these conditions are correctly classified. Beyond those base cases we were able to treat some cases where those conditions weren't present, for example, we're able to treat images with two signs, as long as they are from different colors. Likewise, we're able to also classify some images with poor lighting and with signs not facing the camera. Our solution stops working on extreme cases where the base conditions are not totally present, for example, an image with 4 blue rectangle signs, an image with only half the sign showing, heavy color loss or very small signs are challenging for our model.

9 Future Improvements

Considering the final solution presented, there are some improvements we considered doing but time was an issue. The structure of the project allows for example, to add additional types of signs such as blue circles and triangles. This would be possible by adding additional conditions on the evaluation, filtering for more than 4 edges on the pipeline of blue rectangles and searching for shapes with 3 edges on the pipeline of the red circles/stop signs.

Other improvements that we would apply are the detection of every sign in the case of an image with more than one sign. In the current state on the solution, more than one sign can be classified on an image only if they are of different colors (an image with a red circle, a stop sign and a blue rectangle would be treated and classified with having two signs, a blue rectangle and a red circle/stop sign) due to how our pipeline is setup. The improvement would be to be able to identify all signs, even if they're all of the same type (an image with three blue rectangles would be classified with having three blue rectangles. In the current state it would be only classified in having just one).

To improve the detection of partially hidden signs we could perform angle measurement. In the case of stop signs if we were able to detect half of a octagon based on the angles between the edges and their size we could confidently guess that a stop sign is present. This can also be applied for the squares, where if 3 angles match approximately 90° there is a very high likelihood that there is a blue square present.

10 Annexes

10.1 Pipeline examples

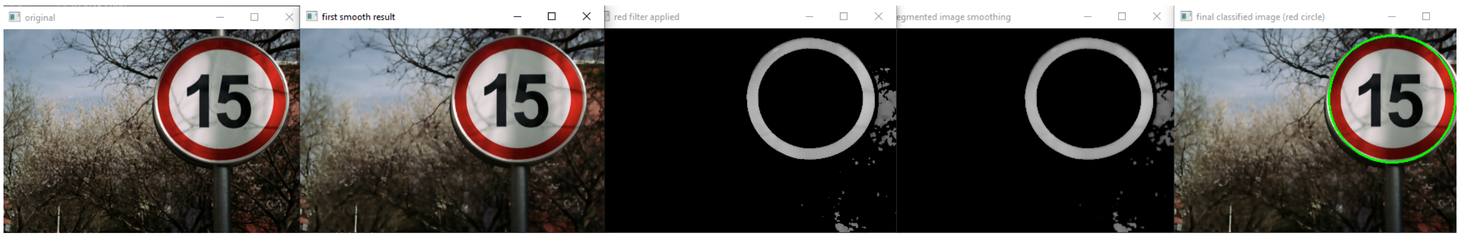


Figure 1: Red circle Pipeline 1

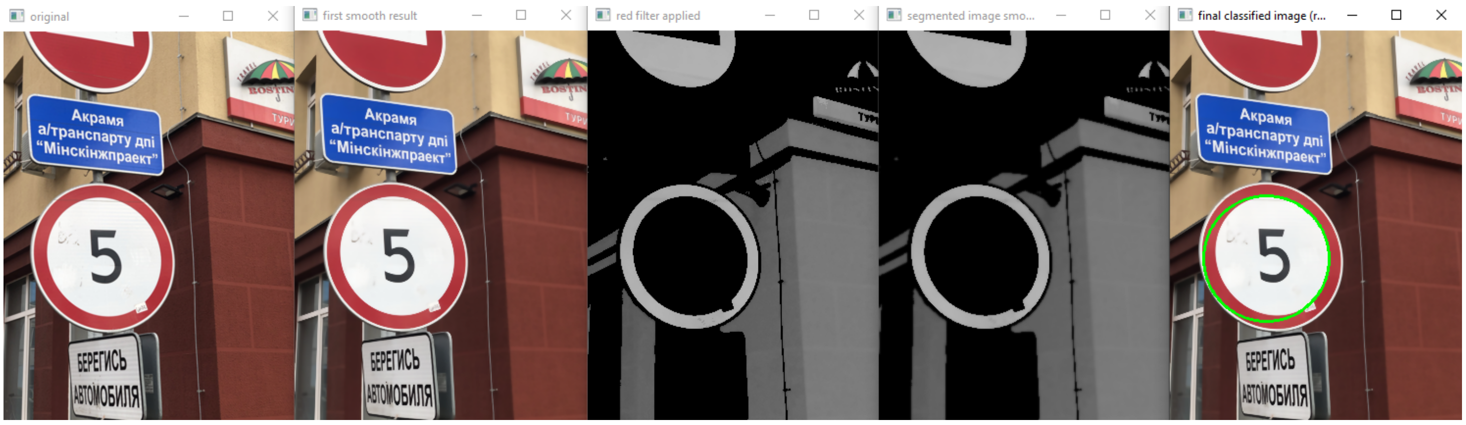


Figure 2: Red Circle Pipeline 2

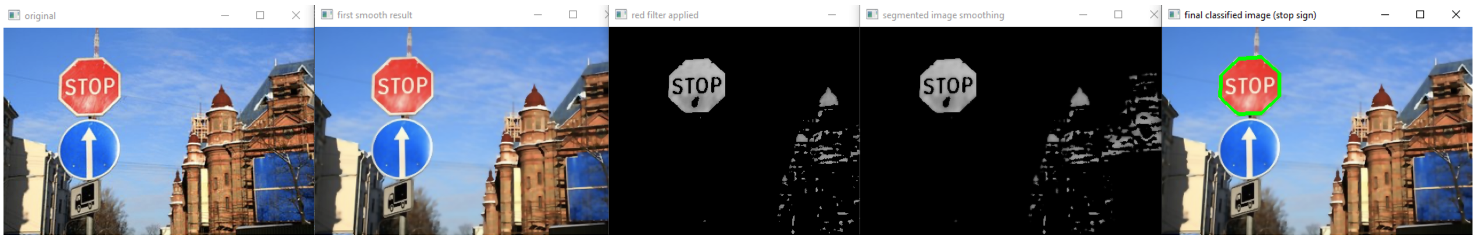
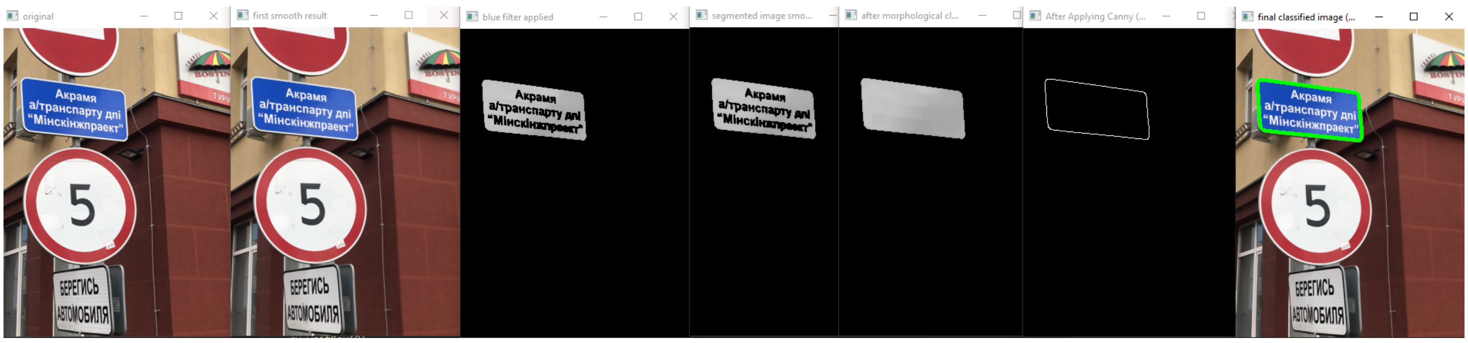


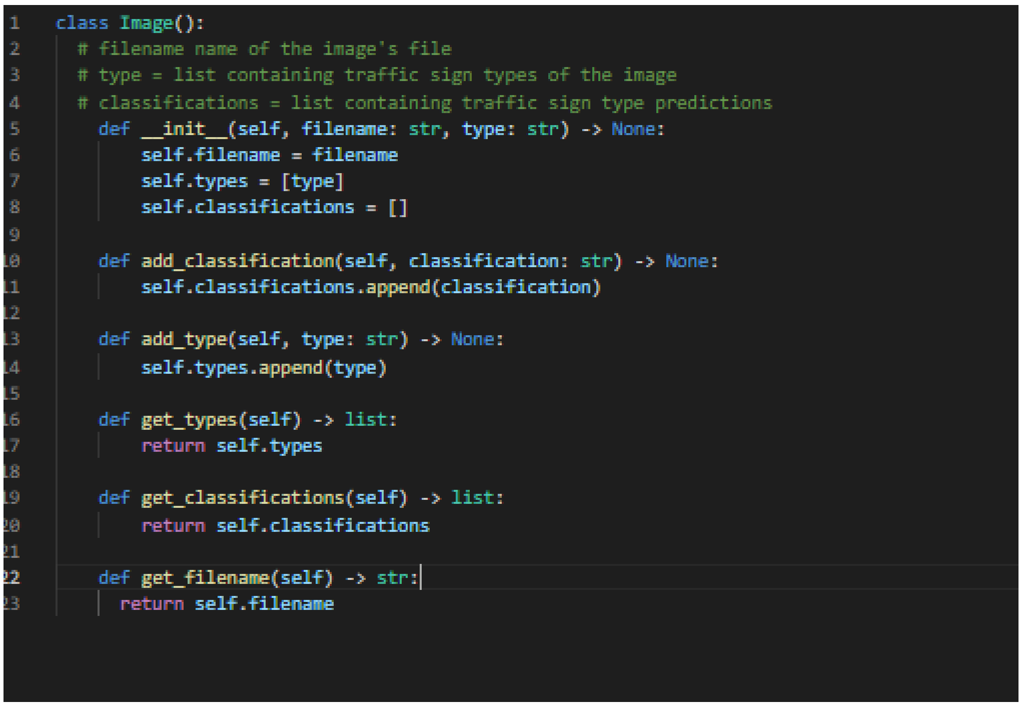
Figure 3: Stop Sign Pipeline



original
first smooth result
blue filter applied
segmented image smo...
after morphological cl...
After Applying Canny (...
final classified image (...
Акрамя а/транспарту дпі "Мінскінжпраект"
5
БЕРЕГИСЬ АВТОМОБИЛЯ

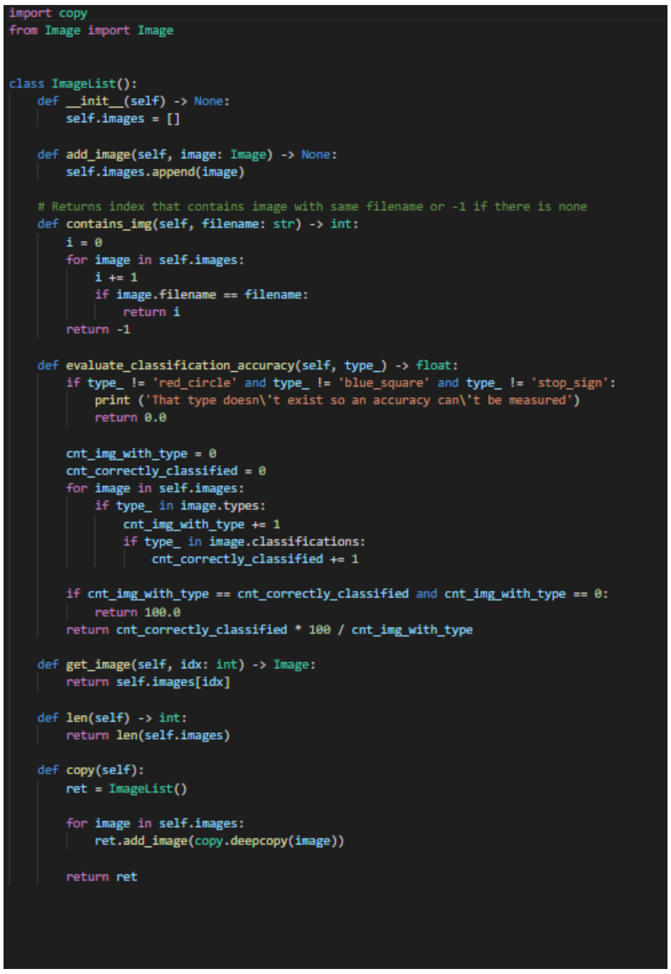
Figure 4: Blue Square Pipeline

10.2 Code



```python
class Image():
    # filename name of the image's file
    # type = list containing traffic sign types of the image
    # classifications = list containing traffic sign type predictions
    def __init__(self, filename: str, type: str) -> None:
        self.filename = filename
        self.types = [type]
        self.classifications = []

    def add_classification(self, classification: str) -> None:
        self.classifications.append(classification)

    def add_type(self, type: str) -> None:
        self.types.append(type)

    def get_types(self) -> list:
        return self.types

    def get_classifications(self) -> list:
        return self.classifications

    def get_filename(self) -> str:
        return self.filename
```

Figure 5: Image class code



```python
import copy
from Image import Image


class ImageList():
    def __init__(self) -> None:
        self.images = []

    def add_image(self, image: Image) -> None:
        self.images.append(image)

    # Returns index that contains image with same filename or -1 if there is none
    def contains_img(self, filename: str) -> int:
        i = 0
        for image in self.images:
            i += 1
            if image.filename == filename:
                return i
        return -1

    def evaluate_classification_accuracy(self, type_) -> float:
        if type_ != 'red_circle' and type_ != 'blue_square' and type_ != 'stop_sign':
            print ('That type doesn\'t exist so an accuracy can\'t be measured')
            return 0.0

        cnt_img_with_type = 0
        cnt_correctly_classified = 0
        for image in self.images:
            if type_ in image.types:
                cnt_img_with_type += 1
                if type_ in image.classifications:
                    cnt_correctly_classified += 1

        if cnt_img_with_type == cnt_correctly_classified and cnt_img_with_type == 0:
            return 100.0
        return cnt_correctly_classified * 100 / cnt_img_with_type

    def get_image(self, idx: int) -> Image:
        return self.images[idx]

    def len(self) -> int:
        return len(self.images)

    def copy(self):
        ret = ImageList()

        for image in self.images:
            ret.add_image(copy.deepcopy(image))

        return ret
```

Figure 6: ImageList class code

Classifying Traffic Signs with OpenCV

For this project we will be trying split the traffic sign dataset into 3 separate classes:

- Stop signs
- Red Circles
- Blue rectangles/squares

Imports

```
import cv2 as cv
import numpy as np
import os, os.path
import shutil
from ImageList import ImageList
from Image import Image
```

Paths and image list

```
dataset_path = 'dataset'
results_path = 'results'

# Results directories reset
shutil.rmtree(results_path + os.sep + 'blue_squares')
shutil.rmtree(results_path + os.sep + 'stop_signs')
shutil.rmtree(results_path + os.sep + 'red_circles')
shutil.rmtree(results_path + os.sep + 'unknowns')
os.mkdir(results_path + os.sep + 'blue_squares')
os.mkdir(results_path + os.sep + 'stop_signs')
os.mkdir(results_path + os.sep + 'red_circles')
os.mkdir(results_path + os.sep + 'unknowns')
```

Useful functions

This function reads the images from the dataset and transfers them to a python list removing possible duplicates.

```
def get_image_list():
    # Saves images into a list containing objects of class Image which
contains traffic sign info
    image_list = ImageList()
    for root, _, files in os.walk('./dataset'):
        for file in files:
            split_file_path = os.path.join(root, file).split(os.sep)
            img_name = split_file_path[-1]
            _type = split_file_path[-2][:-1]
            # Prevent image duplicates
            idx = image_list.contains_img(img_name)
            if idx >= 0:
                image = image_list.get_image(idx)
                image.add_type(_type)
            else:
                image = Image(img_name, _type)
                image_list.add_image(image)

    return image_list
```

The following function takes a gray level picture (segmented_img) and applies several transformations to improve the chances of a sign being detected.

Operations performed:

- Smoothing
- Morphological closing (dilation followed by erosion) with a 5x5 kernel (to blue signs)
- Canny edge detector (to blue signs)

After this contours are searched that may match a blue square or a stop sign. If a contour with more than 8 edges is found then we verify if it is a circle.

This returns the final picture with the contours and the predictions.

```
def process_image(segmented_img, original_img, color, show_images):
    segmented_img = smooth(segmented_img)

    if show_images:
        cv.imshow('segmented image smoothing', segmented_img)

    if color == "blue":
        kernel = np.ones((5,5), np.uint8)
        segmented_img = cv.dilate(segmented_img, kernel, iterations=3)
        segmented_img = cv.erode(segmented_img, kernel, iterations=3)
        if show_images:
            cv.imshow('after morphological closing (blue signs)', 
segmented_img)
        segmented_img = cv.Canny(segmented_img, 150, 200)
        if show_images:
            cv.imshow('After Applying Canny (Blue Signs)', 
segmented_img)

    final, prediction = search_contours(segmented_img, 
original_img.copy(), color)

    if type(prediction) != str: # Prediction is returned as a number 
if found contour has more than 8 edges
        final, prediction = find_circles(segmented_img, 
original_img.copy(), int(prediction))

    return final, prediction
```

For image smoothing we use a bilateral filter

```
# Remove image noise with bilateral filter (better at preserving
edges)
def smooth(img):
    return cv.bilateralFilter(img, 5, 75, 75)
```

The following function increases the brightness of the provided image so that colors can be better identified. Isn't used in pipeline because of worse results

```
# Increases image brightness
def increase_brightness(img, value=30):
    hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
    h, s, v = cv.split(hsv)

    lim = 255 - value
    v[v > lim] = 255
    v[v <= lim] += value

    final_hsv = cv.merge((h, s, v))
    img = cv.cvtColor(final_hsv, cv.COLOR_HSV2BGR)
    return img
```

The segment function looks for the color specified in the argument and creates a black and white mask with the pixels of the desired color set to white.

Red is found at both ends of the HSV hue spectrum so two masks are needed for the red color.

```
#  Segments reds and blues of an image
def segment(img, color):
    img_hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
    if color == 'red':
        # lower mask (0-10)
        lower_red = np.array([0,90,50])
        upper_red = np.array([10,255,255])
        mask0 = cv.inRange(img_hsv, lower_red, upper_red)

        # upper mask (160-180)
        lower_red = np.array([160,90,50])
        upper_red = np.array([180,255,255])
        mask1 = cv.inRange(img_hsv, lower_red, upper_red)

        mask = mask0 | mask1
    elif color == 'blue':
        lower_blue = np.array([90,140,50])
        upper_blue = np.array([130,255,255])
        mask = cv.inRange(img_hsv, lower_blue, upper_blue)

    segmented_img = cv.bitwise_and(img_hsv, img_hsv, mask = mask)
    segmented_img = cv.cvtColor(segmented_img, cv.COLOR_BGR2GRAY)
    return segmented_img
```

This function finds the largest contour in a black and white image. It detects and draws the contours if they match the specified criteria, minimum area, minimum perimeter and maximum perimeter.

```
# Find the contour with the largest area in a gray scale image
def find_largest_contour(segmented_img, original_img):
    contours, _ = cv.findContours(segmented_img,2,1)
    big_contour = []
    max = 0
    for i in contours:
        area = cv.contourArea(i)
        if(area > max):
            max = area
            big_contour = i
    if max > 1000 and len(big_contour) > 10 and len(big_contour) <
1200:
        final = cv.drawContours(original_img, big_contour, -1,
(0,255,0), 3)
    else:
        final = original_img

    return final, big_contour
```

The following function verifies if the segmented image has any white pixels, calls the find_largest_contour function and processes the return.

It counts the number of edges found on the segmented image classifies the sign according to it.

```
# Find a blue square
def search_contours(segmented_img, original_img, color):
    if cv.countNonZero(segmented_img) == 0:
        return original_img, 'unrecognised'
    final, contour = find_largest_contour(segmented_img, original_img)
    if len(contour) > 100:
        peri = cv.arcLength(contour, True)
        approx = cv.approxPolyDP(contour, 0.01 * peri, True)
        if len(approx) > 8: # circle possibly
            return final, cv.contourArea(contour)
        return final, classify_sign(approx, color)

    return final, 'unrecognised'
```

Classifies the sign according to the number of edges present

```
def classify_sign(approx_cnt, color):
    if color == 'blue':
        if len(approx_cnt) == 4:
            return 'blue_square'
    elif color == 'red':
        if len(approx_cnt) == 8:
            return 'stop_sign'
    return 'unrecognised'
```

The following function applies further processing and attempts to find circles in the picture.

```
def find_circles(img, original_img, radius):
    # Reduce white noise and enhance shapes
    kernel = np.ones((3, 3), np.uint8)
    img = cv.dilate(img, kernel, iterations = 3)
    img = cv.erode(img, kernel, iterations = 3)
    #Tolerance
    tol = 1
    circles = cv.HoughCircles(img, cv.HOUGH_GRADIENT, 1, radius * 2 -
tol, param1=100, param2=45, minRadius=5, maxRadius=radius + tol)

    if circles is not None:
        circles = np.uint16(np.around(circles))

        for i in circles[0, :]:
            center = (i[0], i[1])
            r = i[2]
            cv.circle(original_img, center, r, (0, 255, 0), 2)
        return original_img, 'red_circle'
    else:
        return original_img, 'unrecognised'
```

Image list obtained from dataset directory in root

```
img_list = get_image_list()
```

Smoothing applied in different solutions

In order to reduce the amount of unrecognised traffic signs in the dataset images we tried to increase images smoothing until the results started getting worse. Thats why the "amount" of smoothing done in original images for blue segmented image processing is higher than the "amount" used in original images for red segmented images.

Different amount of bilateral (filter) smoothing was applied in original images before segmenting

```
# Counters
cnt_un1 = 0    # Counter for the amount of images unrecognised prior
to using more smoothing if it is unrecognised
cnt_un2 = 0      # Counter for the amount of images unrecognised after
using more smoothing

# Image Lists
img_list1 = img_list.copy() # Image list for low amount of smoothing
```

```
img_list2 =  img_list.copy() # Image list for higher amount of
smoothing

inp = 0
while inp <= 0 or inp > 3:
    inp = int(input('Select solution to be written to the results
directory (1- Normal Smoothing; 2- High Smoothing): '))
```

Normal amount

Smoothing only applied once in original image and then on the segmented one inside process_image function

```
for i in range(0, img_list1.len()):
    # Get img object from img_list
    img_obj = img_list1.get_image(i)

    # Get path of that image -> {type(qualquer um serve)}/{filename}
    img_path = img_obj.types[0] + 's' + os.sep + img_obj.filename

    # Image in the dataset path
    img_path_in_dataset = dataset_path + os.sep + img_path

    # Read img
    img = cv.imread(img_path_in_dataset)

    # Keep an original copy
    original = img.copy()

    # Smooth image
    img = smooth(img)

    # Segment it by filtering the blue and red color for blue and red
signs
    img_blue, img_red = segment(img, 'blue'), segment(img, 'red')

    # Variable to check if any sign is already detected for it not to
be classified as unrecognised in that case
    sign_detected = False

    # Process blue segmented img
    final, classification = process_image(img_blue.copy(), img.copy(),
'blue', False)

    if classification == 'blue_square':
        img_obj.add_classification(classification)
        if inp == 1:
            cv.imwrite(os.path.join(results_path, classification +
                                    's', img_obj.filename), final)
        sign_detected = True
    final, classification = process_image(img_red.copy(), img.copy(),
'red', False)

    if classification == 'stop_sign' or classification ==
'red_circle':
        img_obj.add_classification(classification)
        if inp == 1:
            cv.imwrite(os.path.join(
                results_path, classification + 's', img_obj.filename),
final)
        sign_detected = True

    if sign_detected == False:
        cnt_un1 += 1
        img_obj.add_classification(classification)
        if inp == 1:
            cv.imwrite(os.path.join(
                results_path, 'unknowns', img_obj.filename), final)
```

Higher amount

Smoothing applied more times in original image if normal smoothing didn't detect any traffic signs

```
for i in range(0, img_list2.len()):
    # Get img object from img_list
    img_obj = img_list2.get_image(i)

    # Get path of that image -> {type(qualquer um serve)}/{filename}
    img_path = img_obj.types[0] + 's' + os.sep + img_obj.filename

    # Image in the dataset path
    img_path_in_dataset = dataset_path + os.sep + img_path

    # Read img
    img = cv.imread(img_path_in_dataset)

    # Keep an original copy
    original = img.copy()

    # Smooth image
    img = smooth(img)

    # Segment it by filtering the blue and red color for blue and red
signs
    img_blue, img_red = segment(img, 'blue'), segment(img, 'red')

    # Variable to check if any sign is already detected for it not to
be classified as unrecognised in that case
```

```
    sign_detected = False

    # Process blue segmented img
    final, classification = process_image(img_blue.copy(), img.copy(),
'blue', False)

    # Process first iteration result
    if classification == 'blue_square':
        img_obj.add_classification(classification)
        if inp == 2:
            cv.imwrite(os.path.join(results_path, classification +
                                    's', img_obj.filename), final)
        sign_detected = True
    elif classification == 'unrecognised':  # if no sign detected do
more smoothing
        temp = smooth(smooth(smooth(img_blue)))
        final, classification = process_image(temp, img.copy(),
'blue', False)
        if classification == 'blue_square':
            img_obj.add_classification(classification)
            if inp == 2:
                cv.imwrite(os.path.join(results_path, classification +
                                        's', img_obj.filename), final)
            sign_detected = True

    # Process red segmented img
    final, classification = process_image(img_red.copy(), img.copy(),
'red', False)

    # Process first iteration result
    if classification == 'stop_sign' or classification ==
'red_circle':
        img_obj.add_classification(classification)
        if inp == 2:
            cv.imwrite(os.path.join(
                results_path, classification + 's', img_obj.filename),
final)
        sign_detected = True
    elif classification == 'unrecognised':  # if no sign detected do
more smoothing
        temp = smooth(img_red)
        final, classification = process_image(temp, img.copy(), 'red',
False)
        if classification == 'stop_sign' or classification ==
'red_circle':
            img_obj.add_classification(classification)
            if inp == 2:
                cv.imwrite(os.path.join(results_path, classification +
                                        's', img_obj.filename), final)
            sign_detected = True
    if sign_detected == False:
        cnt_un2 += 1
        img_obj.add_classification(classification)
        if inp == 2:
            cv.imwrite(os.path.join(
                results_path, 'unknowns', img_obj.filename), final)
```

Amount of unknowns (no traffic signs detected) and accuracy for each traffic sign type. This accuracy is calculated in a method of the ImageList class by dividing the amount of correct classifications by the amount expected.

```
print('==========No Traffic Signs Detected==========')
print('Normal amount of smoothing: ' + str(cnt_un1))
print('Higher amount of smoothing: ' + str(cnt_un2))
print('==========Detection Accuracy for Red Circles==========')
print('Normal amount of smoothing: ' +
str(img_list1.evaluate_classification_accuracy('red_circle')))
print('Higher amount of smoothing: ' +
str(img_list2.evaluate_classification_accuracy('red_circle')))
print('==========Detection Accuracy for Blue Squares==========')
print('Normal amount of smoothing: ' +
str(img_list1.evaluate_classification_accuracy('blue_square')))
print('Higher amount of smoothing: ' +
str(img_list2.evaluate_classification_accuracy('blue_square')))
print('==========Detection Accuracy for Stop Signs==========')
print('Normal amount of smoothing: ' +
str(img_list1.evaluate_classification_accuracy('stop_sign')))
print('Higher amount of smoothing: ' +
str(img_list2.evaluate_classification_accuracy('stop_sign')))
```

Results

The results are all showcased with the respective contours on the results directory.

Pipeline Examples

Using an image with a red circle sign

```
img_path = dataset_path + os.sep + 'red_circles' + os.sep +
'road163.png'
img = cv.imread(img_path)
cv.imshow("original",img)

img=smooth(img)
cv.imshow("first smooth result",img)

img_red=segment(img,'red')
cv.imshow("red filter applied",img_red)

final, classification = process_image(img_red.copy(), img.copy(),
'red', True)
if classification == 'stop_sign' or classification == 'red_circle':
    cv.imshow("final classified image (red circle)",final)
elif classification=='unrecognised':
    cv.imshow("second smooth result", final)
    temp=smooth(img_red)
    final, classification = process_image(temp, img.copy(), 'red')
    cv.imshow("final classified image after further smoothing(red
circle)",final)


cv.waitKey(0)
cv.destroyAllWindows()
```

Using an image with blue square sign

```
img_path = dataset_path + os.sep + 'blue_squares' + os.sep +
'road163.png'
img = cv.imread(img_path)
cv.imshow("original",img)

img=smooth(img)
cv.imshow("first smooth result",img)

img_red=segment(img,'blue')
cv.imshow("blue filter applied",img_red)

final, classification = process_image(img_red.copy(), img.copy(),
'blue', True)
if classification == 'blue_square':
    cv.imshow("final classified image (blue square)",final)
elif classification == 'unrecognised':  # if no sign detected do more
smoothing
        temp = smooth(smooth(smooth(img_blue)))
        final, classification = process_image(temp, img.copy(),
'blue', False)
        if classification == 'blue_square':
            img_obj.add_classification(classification)

cv.waitKey(0)
cv.destroyAllWindows()
```

Using an image with a stop sign

```
img_path = dataset_path + os.sep + 'stop_signs' + os.sep +
'road78.png'
img = cv.imread(img_path)
cv.imshow("original",img)

img=smooth(img)
cv.imshow("first smooth result",img)
```

```
img_red=segment(img,'red')
cv.imshow("red filter applied",img_red)

final, classification = process_image(img_red.copy(), img.copy(),
'red', True)
if classification == 'stop_sign' or classification == 'red_circle':
    cv.imshow("final classified image (stop sign)",final)
elif classification=='unrecognised':
    cv.imshow("second smooth result", final)
    temp=smooth(img_red)
    final, classification = process_image(temp, img.copy(), 'red')
    cv.imshow("final classified image after further smoothing(stop
sign)",final)


cv.waitKey(0)
cv.destroyAllWindows()
```