ENPM 673
Traffic Sign Recognition

Members- Akash Atharv
- Rishabh Choudhary
- Sanchit Gupta

Instructions to run the code

Import the following python 3 libraries in the classification.py file

1. Import cv2
2. Import numpy as np
3. Import os
4. From PIL import Image
5. From skimage import feature
6. From sklearn import svm

Import the following python 3 libraries in the project6_demo.py file

1. Import cv2
2. Import numpy as np
3. Import os
4. Import imutils
5. From PIL import Image
6. Import classification.py as train

In order to run the code, keep the dataset in the same folder as the files with the training files in train_set folder and testing files in test_set folder.

The Google drive link is as given below :
https://drive.google.com/drive/folders/1BjKon6kIaPNvLJ8uRYxROiM_x93OodLu?usp=sharing

Section 1: Introduction

In this project, we have implemented the traffic sign recognition. Here we have detected red and blue signs and identified them from a collection of pre-defined images. After identifying the traffic sign, we paste the image of the identified sign next to the detected sign on the frame.

We have used color segmentation to separate out red and blue color from the frame, then used MSER to detect the regions with the defined area ranges. The output regions of the same are identified using a pre-trained SVM classifier.

The SVM classifier takes in a vector of hog feature images of size 64x64 and outputs the detected label.

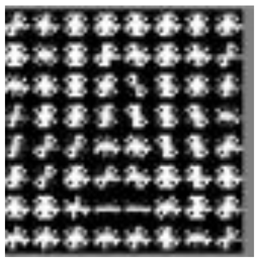
Note: The complete dataset provided was not used, certain folders were taken out and the number of examples from certain traffic signs were reduced to maintain a balance among different data sets.

Section 2: Pipeline

Part 1: Train the SVM classifier

The python file classification.py is used to train and test the classifier. Here the images are read from the train_set folder which has images segregated based on the labels. These images are read and converted from ppm format to np.array() format. This image is then resized to 64x64 shape and converted to a gray scale image.

The reshaped image is then fed into the hog feature detector, this helps to convert the input image to feature vector which can then be used to train any classifier. The outputs of the above steps are shown below:



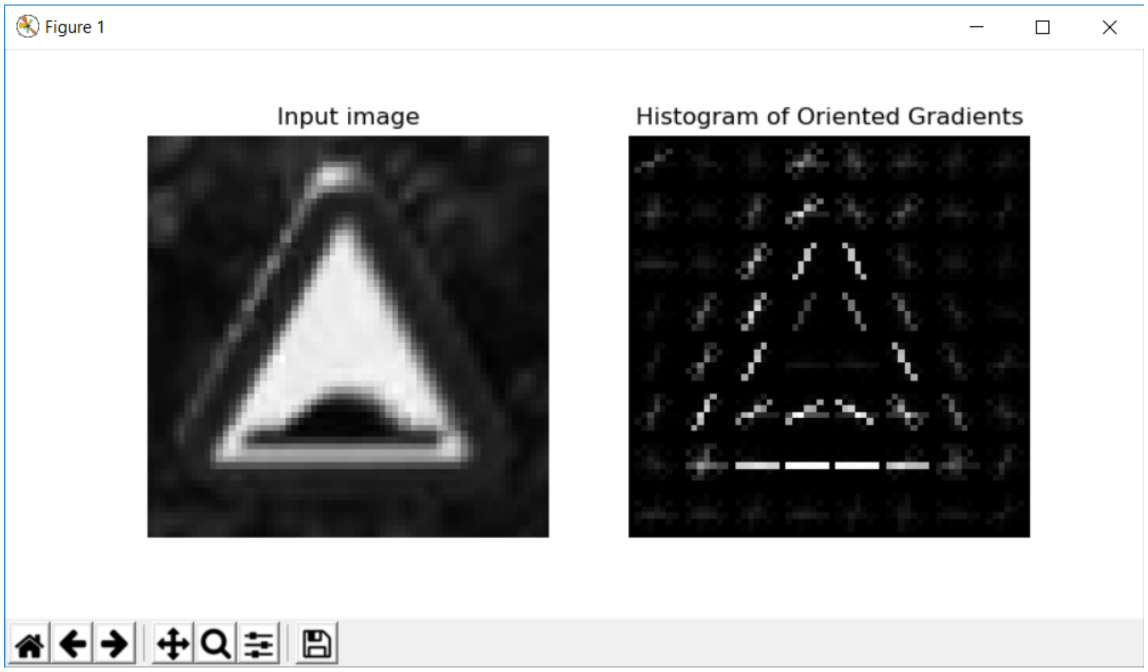


Fig: input training image and it's corresponding Hog feature image

Once the hog features have been detected we store all the hog images in a list with all the corresponding labels and give the same to the SVM classifier. The classifier returns a trained model which can then be tested using the predict function on the test images.

We trained red and blue symbols separately because of the difference in the hog parameters for both. The models were tested on the test images from the given set, these images are also prepared in the similar manner as the trained images, first converted to gray images, resized and then given to Hog feature detector. The following results were obtained:

```
Percentage accuracy:  98.84526558891456
```

$\rightarrow$ For Red

```
Percentage accuracy:  92.46119733924611
```

$\rightarrow$ For Blue

To enhance the results, we trained few of the negative images so that they won't be identified. We took images of blue cars, red building and assigned a common label (100) to them. Now when the predictor outputs the identified image having a label 100, then nothing is plotted.

Example of a false positive image

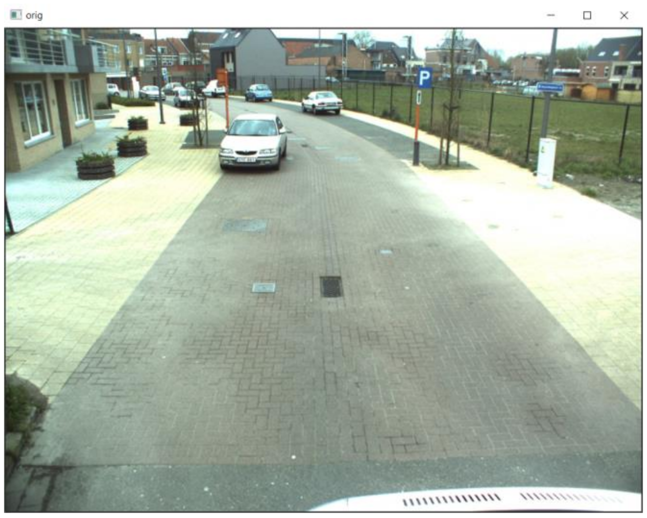
Once the satisfactory results were obtained from models for both red and blue.

The code was saved and was imported in a separate file where the input frame images were read.

Part 2: Reading and preparing the frame images

Part 2A- For Blue and Red Detection:

The input frame is read and is given to the detect blue, detect red functions. The detect blue and red the function first converts the image from BGR format to YUV to perform histogram equalization after which is converts the equalized image back to BGR. This is done to equally divide the color pixel values through out the image.



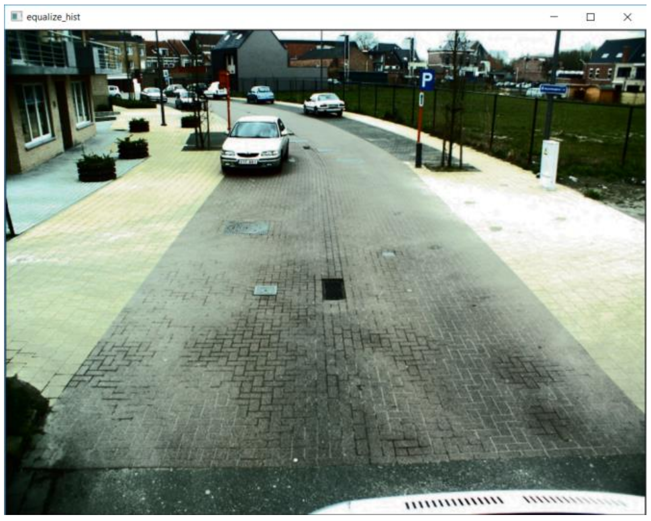


Fig: Frame image before and after equalization

The image is then converted from BGR format to HSV for better color segmentation. Our aim is to separate out red and blue regions from the image which is why HSV color space is better. We then define our mask ranges to separate out the color.

For Blue:

Lower Mask values: [94, 127, 20]

Upper Mask values: [126, 255, 200]

For Red:

Because the red appears in 2 different regions in the HSV color space, we need 2 separate ranges for it. The same are:

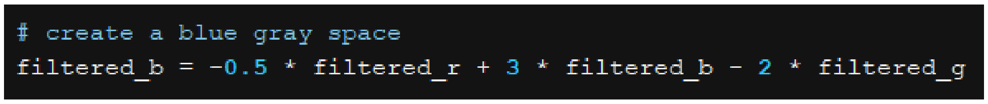
Lower Mask 1 values: [0, 70, 60]

Upper Mask 1 values: [10, 255, 255]

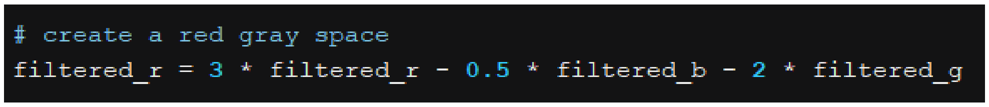
Lower Mask 2 values: [170, 70, 60]

Upper Mask 2 values: [180, 255, 255]

Any pixel in the HSV image having pixel values between the above defined range is colored and all the other regions are colored black. This image is then separated into it’s 3 channels, these 3 channels are median filtered separately and are used to define a custom gray image for better identification of the red and blue regions. These gray images are defined on the bases of the pixel values of different channels.



```
# create a blue gray space
filtered_b = -0.5 * filtered_r + 3 * filtered_b - 2 * filtered_g
```

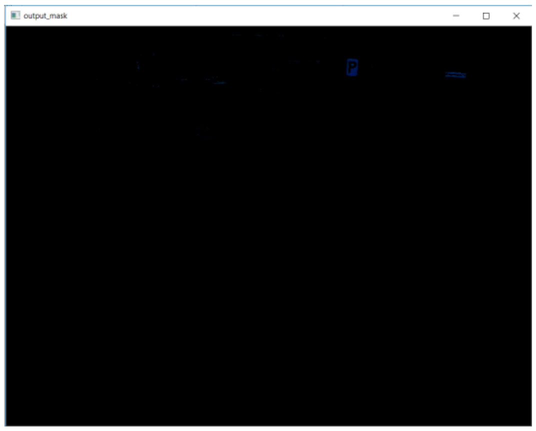


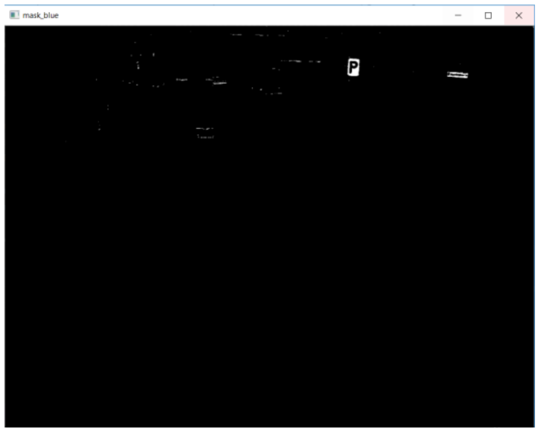
```
# create a red gray space
filtered_r = 3 * filtered_r - 0.5 * filtered_b - 2 * filtered_g
```

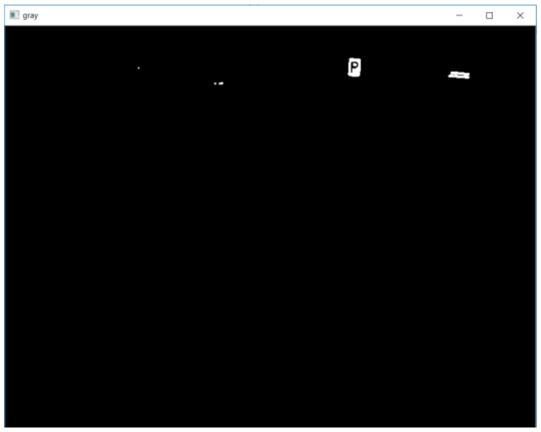
Fig: The gray image parameters

These gray images are then given to the mser.detectRegions() function. This function returns the detected regions in the input image which are within certain parametric limits. These regions are then given to the convex hull function and the output are stored in a list.

The hulls detected from the MSER are then drawn on a blank black image and are smoothed and prepared using different functions like “erosion”, “dilate”, ”morphologyEx”, the red and the blue channel of the output image of all these functions is then thresholded and given to the find contours function. These contours are sorted based on the area and for each frame top 3 contours are taken.







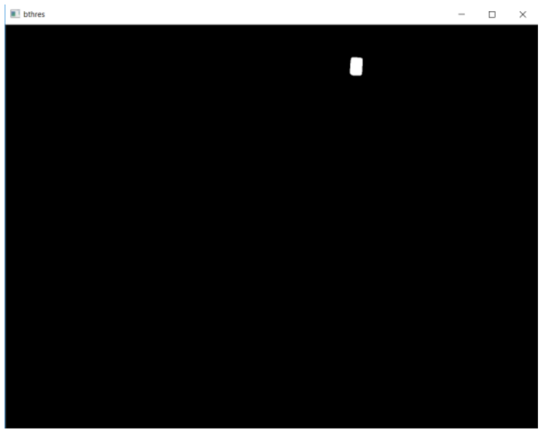


Fig: Blue mask of the frame, performing operations on the mask, after thresholding result

Part 2B- Find the contours that we need:
Once we have the top 3 contours from each image, we still have to identify which of these corresponds to the traffic sign, to do this we use our previous knowledge of shape and size of the traffic signs. For red signs we found that all the signs appear on the right half of the image which is why we restricted finding the contours in that area and similarly we did for blue and neglected the bottom 1/3rd of the image.

The second check we implemented was for the position and the shape of the contour. These values were found using the function cv2.boundingRect(), which returns x_position, y_position, height, width of the contour. All the contour with height less than a certain value were discarded.

The third check was using the aspect ratio. If the aspect ratio for the detected contour went above or below a certain value, it was also neglected.

The contour which are left, a rectangle was made for each of these on a mask image using the previously calculated x_position, y_position, height, width. The regions where the rectangle was plotted were taken out from the original image and were given to the predict function of the previously trained SVM classifier.

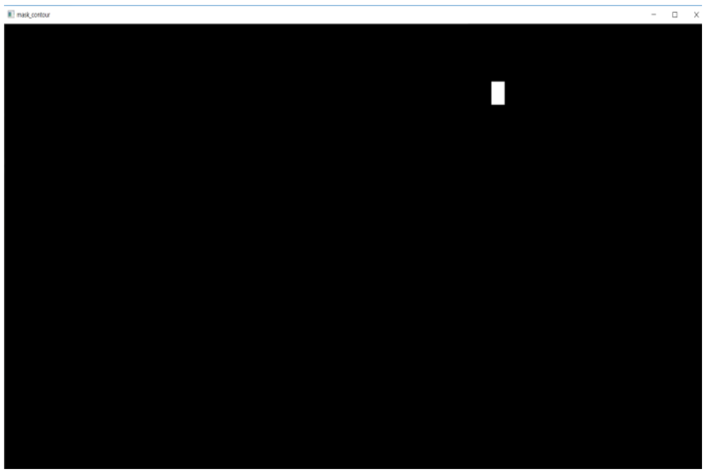


Fig: Mask to crop out the region from the original image and the cropped out image

The classifier returns the label it identified and also the probability with which it gives this result. This result is further used to limit the miss-detections. Whenever the probability is below a certain value, we drop that contour and not plot it.

Part 2C- Plot the bounding rectangle and the identified image on the left

Once the prediction is done, the contours and the label are stored in a list and is returned from the functions. Here we again use the boundingRect() function to identify the x and y values of the contours, then plot a rectangle on the contour and also replace the left 64x64 pixel in the image frame with the identified traffic sign image.

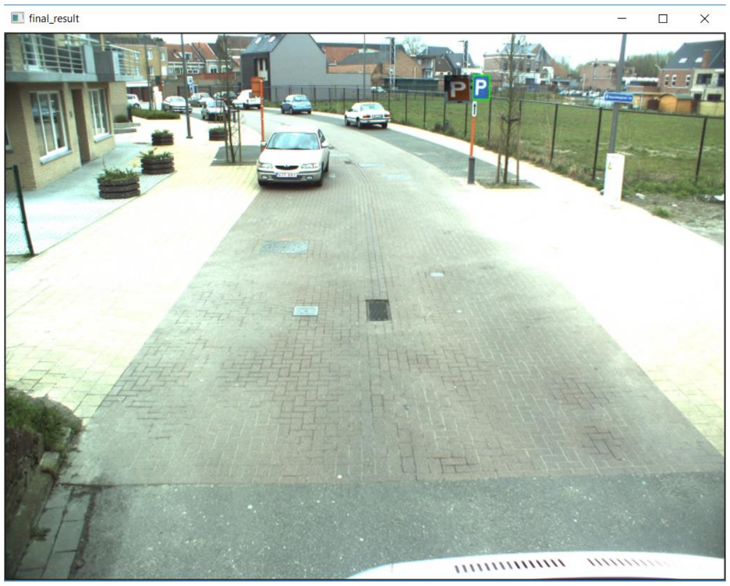
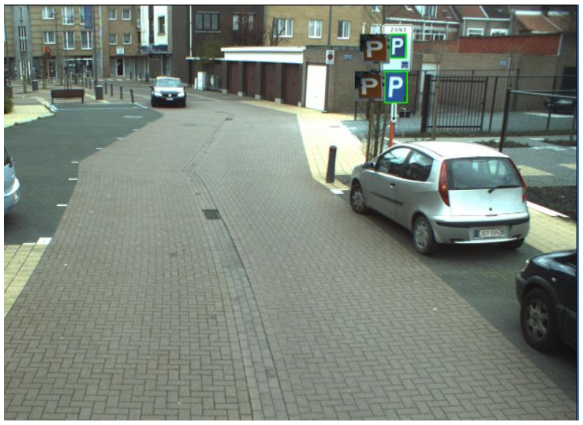
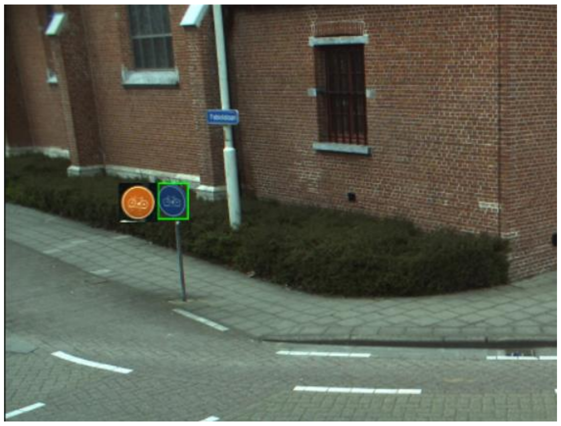
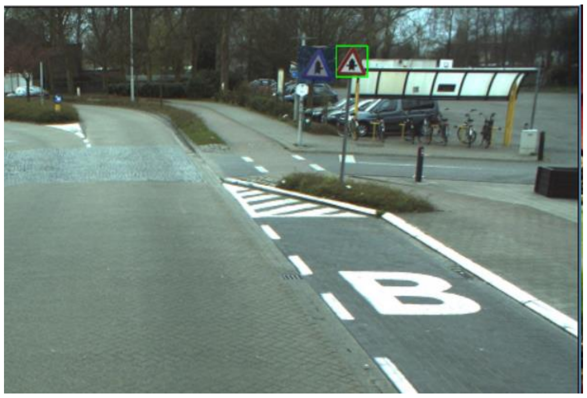
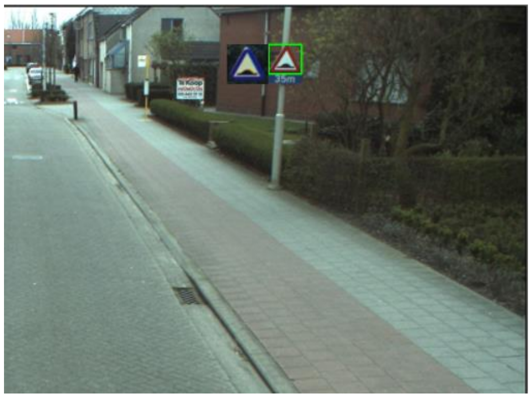


Fig: Final result of the same frame

One image of each of the traffic sign is stored in a dictionary with the key being the label number of it.

The same is done for both red and blue contours using a for loop and the results are plotted on the same image which is then displayed in the end.

Some images of the final result:

Section 3 : Challenges faced

- **Hog Parameters**: The hog feature detector function takes in many inputs which can influence the results quite a lot, because of this we were not getting good enough testing accuracy (87%) initially and had to read and tweak these values to increase the accuracy and get a better model.
- **Mask Parameters**: To segment out blue and red using HSV, the values for the traffic signs was quite confusing, we had to try and tweak this value quite a lot before finalizing the values.
- **Placing the identified image:** To place the identified image on the left of the actual sign, we faced some issues because sometimes there was not enough space on the left to place the image. We had to keep a check of the same.

- **False Positive:** We were getting a lot of false positives on the car and the red building, to remove these, we trained some negative images of these areas and predicted these values. Whenever these were encountered, nothing was plotted on the frame image.

Section 4: Reference

1. https://docs.opencv.org/3.3.1/d5/d69/tutorial_py_non_local_means.html
2. https://stackoverflow.com/questions/31647274/is-there-any-function-equivalent-to-matlabs-imadjust-in-opencv-with-c
3. https://github.com/opencv/opencv/blob/master/samples/python/mser.py
4. http://www.micc.unifi.it/delbimbo/wp-content/uploads/2011/03/slide_corso/A34%20MSER.pdf
5. https://www.learnopencv.com/handwritten-digits-classification-an-opencv-c-python-tutorial/