Using Logistic Regression and OpenCV Library for Stop Sign Detection in Images

William Argus[1]
Department of Electrical and Computer Engineering, UCSD
wargus@eng.ucsd.edu

***Abstract*— This paper presents an approach to stop sign detection that uses logistic regression for color classification and shape analysis for stop sign detection. In this work, the idea presented in Section II is implemented in Python with the final goal being to identify stop signs in previously unseen images. The result showed that this was generally successful with most images, however the program struggled with low resolution images and images with small stop signs.**

I. INTRODUCTION

Color classification is a problem that involves using pre-existing and labeled data to find the determine characteristics about both the desired and undesired colors, in order to develop a mathematical model of the different colors, known as classes, for use in identifying colors in previously unseen images. The approach employed by this author in this paper is Logistic Regression, which the author uses to identify the values of hyper-parameters, which can then be used to classify any pixel, in this case as either red or not red, since the main concern is stop sign identification.

The second step to this solution involves shape analysis in order to take areas of an image classified as red and make a determination as to if the areas are a stop sign or not. In doing this shape analysis, it has to be taken into account that these are real images of stop signs taken and that the red regions are determined by a classifier. What this means is that it is not as simple as looking for an octagon in the red areas, as the shape analysis often featured obscured or poorly classified stop signs. This can lead to the shape analysis section of this project relying heavily on OpenCV functions to check features of the red regions for similarity with stop signs, and attempting to strike a sensible balance in the finding of stop signs, and discarding of false positives. This is detailed in the Technical Approach section of this paper.

II. PROBLEM FORMULATION

A. Classifying Problem

In order to solve the problem of classifying each pixel in the image, the student elected to use a discriminate model. This was because it is not necessary to generate new data points, merely to classify the data given to the program in the form of images. The classes are therefore defined to be red and not red, for which $y = 1 and y = -1$, respectively. Each pixel can be classified for a descriminative, maximum likelihood classifier with the following equation.

$$y = argmax_y p(y|x_*, \omega_{MLE}) \tag{1}$$

In the above equation, ω_{MLE} is the vector $\in R^d$, containing the parameter weights optimized for best performance. Note that here, d, the dimension of data for each pixel, is 3, as in the RGB, YUV, and HSV color spaces. The term x_* from dataset $D \in R^{nxd}$, represents the pixel being classified in the equation.

B. Optimizing ω

In order to optimize the term ω_{MLE}, the following equation is used.

$$\omega^* = argmax_\omega p(y|X, \omega) \tag{2}$$

As mentioned above, X is the dataset of pixels, and y is a vector, each vertical entry of which corresponds to a pixel in X. This vector y can be reshaped into an array with the same dimenions as the image and has the ability to function as a mask for the classifier. This will be discussed more in the following section.

III. TECHNICAL APPROACH

This section details the approach used to solve the problems listed in the problem formulations. It will describe first the equations used to solve the classification problem and then the code used to implement this solution. It will then describe the techniques used to filter the results of the classification such that stop signs and only stop signs are identified by the final bounding boxes given as the program output.

A. Logistic Regression

The first step to detecting stop signs is to classify each in the image as either red or not red. The approach used for this was to use Logistic Regression to obtain a model for the probability of the classes below in equation (2).

$$p(y|X, \omega) = \prod_{i=1}^{n} \sigma(y_i x_i^T \omega) \tag{3}$$

Where y is the class label, $y = 1$ corresponds to red and $y = -1$ corresponds to not red, x_i is the pixel

being examined at the given moment, and ω is the model parameters. The model parameters are found by iterating equation (3) until it converges to the values of ω.

$$\omega_{MLE}^{(t+1)} = \omega_{MLE}^{(t)} + \sum_{i=1}^{n} y_i x_i (1 - \sigma(y_i x_i^T \omega_{MLE}^{(t)}) \quad (4)$$

Where σ is the sigmoid function in equation (5), and alpha is the step size, for this project, 0.01 was used. This was small enough to accurately find the minimum of the gradient descent equation (6), yet large enough that the program ran sufficiently fast. It should be noted that alpha was found experimentally in this case.

$$\sigma(z) = \frac{1}{1 + exp(-z)} \quad (5)$$

$$0 = \nabla_\omega(-log[p(y|X, \omega)]) \quad (6)$$

Once equation (4) is solved iteratively for ω^*, the following decision boundary can be used to determine the class of any pixel, X_*^T, that is multiplied with the optimized parameters ω^*, seen in equation (7).

$$y_* = \begin{cases} 1 & \text{if } X_*^T * \omega^* \geq 0 \\ -1 & \text{if } X_*^T * \omega^* < 0 \end{cases} \quad (7)$$

]

B. Python Implementation of Logistic Regression

The first step to implementing logistic regression is to gather a training data set. In order to do that, 100 training images were selected from the images given with the project. These images were hand cropped to create regions of red stop signs, and regions without any stop signs. These images were then placed into a folders called RedPixels and notRedPixels.

A Python script was written to go through each directory and extract 1 image at a time. The pixels from each image were then stored in a 3 column array, with each column representing 1 of the RGB values for each pixel. Then labeling arrays for red and not red pixels was created with the red pixels being labeled 1 and the not red pixels being labeled -1. The arrays of pixel values for red and not red and their corresponding labeling arrays were concatenated together to create 1 array of training pixels and a corresponding array of a label or either 1 or -1 for each pixel.

A second python script was then written to find the value for optimized ω: ω^*. This python script imported the numpy arrays of training data and corresponding labels and used the following equation, seen previously to iteratively find ω^*.

$$\omega_{MLE}^{(t+1)} = \omega_{MLE}^{(t)} + \sum_{i=1}^{n} y_i x_i (1 - \sigma(y_i x_i^T \omega_{MLE}^{(t)}) \quad (8)$$

Due to this being this student's first time using Python, this student was unable to get the numpy matrix multiplication

	0	1	2
0	-1.73625e+06	-1.75239e+06	401287
1	-1.20906e+06	-1.43386e+06	2.47985e+06
2	-1.77016e+06	-2.09984e+06	1.74076e+06
3	-1.73755e+06	-2.07006e+06	1.79785e+06
4	-1.7104e+06	-2.04537e+06	1.84355e+06
5	-1.68775e+06	-2.02501e+06	1.88008e+06
6	-1.66867e+06	-2.00817e+06	1.90949e+06
7	-1.65252e+06	-1.99438e+06	1.93306e+06
8	-1.63876e+06	-1.9831e+06	1.95196e+06
9	-1.62698e+06	-1.97389e+06	1.96707e+06
10	-1.61678e+06	-1.96635e+06	1.97918e+06
11	-1.60794e+06	-1.96022e+06	1.98875e+06
12	-1.60022e+06	-1.95528e+06	1.99624e+06
13	-1.59338e+06	-1.95129e+06	2.00209e+06
14	-1.58731e+06	-1.94812e+06	2.00652e+06
15	-1.58181e+06	-1.94556e+06	2.0099e+06
16	-1.5768e+06	-1.94353e+06	2.01239e+06
17	-1.57225e+06	-1.94199e+06	2.01404e+06
18	-1.56801e+06	-1.9408e+06	2.0151e+06
19	-1.564e+06	-1.93984e+06	2.01577e+06
20	-1.56024e+06	-1.93915e+06	2.01599e+06
21	-1.55664e+06	-1.93864e+06	2.0159e+06
22	-1.5532e+06	-1.9383e+06	2.01552e+06
23	-1.54988e+06	-1.93808e+06	2.01495e+06
24	-1.54664e+06	-1.93795e+06	2.01422e+06
25	-1.54346e+06	-1.93789e+06	2.01339e+06
26	-1.54034e+06	-1.93788e+06	2.01245e+06
27	-1.53728e+06	-1.93794e+06	2.0114e+06
28	-1.53426e+06	-1.93803e+06	2.0103e+06
29	-1.53127e+06	-1.93815e+06	2.00913e+06
30	-1.52829e+06	-1.93827e+06	2.00796e+06
31	-1.52535e+06	-1.93842e+06	2.00674e+06
32	-1.52241e+06	-1.93858e+06	2.0055e+06
33	-1.51949e+06	-1.93875e+06	2.00425e+06
34	-1.51657e+06	-1.93892e+06	2.00298e+06
35	-1.51369e+06	-1.93912e+06	2.00166e+06
36	-1.51083e+06	-1.93934e+06	2.00031e+06
37	-1.50797e+06	-1.93955e+06	1.99897e+06
38	-1.50511e+06	-1.93976e+06	1.99762e+06
39	-1.50225e+06	-1.93996e+06	1.99628e+06
40	-1.49941e+06	-1.94017e+06	1.99492e+06
41	-1.49658e+06	-1.94039e+06	1.99355e+06
42	-1.49376e+06	-1.94061e+06	1.99217e+06
43	-1.49095e+06	-1.94084e+06	1.99076e+06
44	-1.48814e+06	-1.94106e+06	1.98937e+06
45	-1.48536e+06	-1.94129e+06	1.98794e+06
46	-1.48257e+06	-1.94153e+06	1.98651e+06
47	-1.47978e+06	-1.94175e+06	1.98511e+06
48	-1.47699e+06	-1.94197e+06	1.9837e+06
49	-1.47421e+06	-1.94218e+06	1.9823e+06

Fig. 1. Table showing the successive value of ω^* after 50 iterations

functioning correctly such that the summation could be computed using matrix operations. The alternate approach taken was to iterate through all the training data points for each iteration, finding the value of the quantity $y_i x_i (1 - \sigma(y_i x_i^T \omega_{MLE}^{(t)})$ at each data point, and then summing them in the loop. This student concedes that this was not an optimal way to write the program, however due to this student's lack of experience, it was the only route this student was able to take. Nonetheless, it did work as intended.

The number of iterations necessary to achieve converge for the values of ω^* was found experimentally, by performing trials with 5, 15, 25, and finally 50 iterations. After 50 iterations, it was determined that the delta value, the difference between the previous value the the new value of ω^* was so small, that the gradient descent had reached a minima. The results of these 50 iterations can be seen in Fig 1.

It should be noted that the student also tried first convert-

ing the BGR images into HSV images and normalizing the brightness across images and converting back to BGR, but this was found to have little affect on the effectiveness of the pixel classifier so it was not used in the final version of the code.

C. Python Implementation of Image Mask

Once the parameter values are found, the next step is to segment the image. This was a core part of the assignment, as the directive specifically called for implementing the **segmentimage** function. This student implemented said function in the following manner.

The image passed to the function was reshaped reshaped into an array with 3 columns, each representing 1 of the BGR values, and then matrix multiplied with the vector containing the values of ω^*. The result was the creation of a vector that contained values for each pixel such that the last step in classification, as per equation (7), was to simply make all values in the vector greater than or equal to zero 1, representing red pixels, and all values less than -1, representing not red pixels, and fully classifying the image.

Creation of the mask was done by changing all entries in the vector that were equal to 1, to 255, hence making all red pixels show up as white in the mask, and all entries in the vector that were equal to -1, 0, making all not red pixels blacked. The vector was then rehaped with the **np.reshape** function and then transposed with the **np.transpose** function to make it into an array with the same dimensions as the image. This array was returned by the function.

D. Python Implementation of Stop Sign Detection

In order to detect stop signs, the first step was to pass the image to the **segmentimage** function, in order to get the mask of the image showing all red regions. The mask was then blurred with the **cv2.GaussianBlur** function to help smooth the edges of the mask, and then threshold-ed back to black and white and returned to black and white BGR image with the **cv2.threshold** and **cv2.cvtColor** functions, respectively.

In order to define contours surrounding all the areas of the mask detected as red, the **cv2.findContours** function was called. Once these contours were found, all that remained was to determine which were the shape of stop signs and which were not. First, but the largest 3 contours by area were discarded with the sort function. This was done as there were many instances of small false detections consisting of only a few pixels, whereas the stop signs were generally large in area. The idea to keep 3 and not some other number of contours was developed experimentally, and 3 was found to be successful for the widest range of test cases. Next, the number of lines making up the contour was required to be between 6 and 14. Ideally a stop sign should have exactly 8, but due to obscured stop signs, and jagged masks of stop signs, it was necessary to widen the range, otherwise the program would fail to identify some stop signs. Next, each contour was had an ellipse fitted to it, and if the ratio of the minor axis to the major axis was smaller than 0.75993421, that contour was discared as not a stop sign. The value of the ratio was between major and minor axis was also found experimentally through testing to be a good compromise between a square contour of a stop sign, and the rectangular contour of a partially obstructed stop sign. Finally, as one last check, the area of the stop sign was required to be larger than one one-thousandth the area of the image as a whole. This was to account for cases where there was no stop signs, but a bunch of small groupings of red pixels that occasionally they were misidentified as stop signs.

If all of the checks were passed for a contour, the coordinates of its corners were appended in a list to the list of bounding boxes for this image. This list of lists was then sorted into the correct order per the spec with the sort function, using lambda as the key, before being returned by the function.

IV. Results

This section showcases several of the test cases that the completed program was put through and the success and failures are discussed. For all images the segmentation mask is displayed, so that the reader can get a good idea as the effectiveness of the color classifier. Then, for images where stop signs were detected, a bounding box is displayed, as this is easier to visually see than simply the printed out version of the bounding box. For images where stop signs were not found, the contours are displayed, so the reader can get an idea of what areas were classified as red, yet rejected by the shape analysis portion of the program. The results will be displayed first, then discussed in the subsection after, as it makes the formatting with the images work much better.

A. Results displayed

Fig. 2. Case 1

Fig. 3. Case 1, Classification Mask

Fig. 6. Case 2, Classification Mask

Fig. 4. Case 1, Stop Sign Bounding Box

Fig. 7. Case 2, Stop Sign Bounding Box

Fig. 5. Case 2

Fig. 8. Case 3

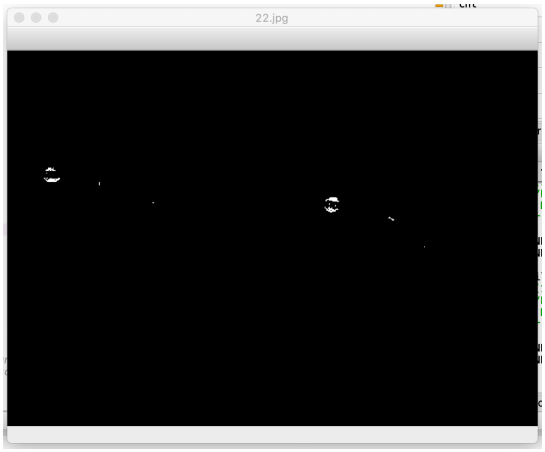

Fig. 9. Case 3, Classification Mask

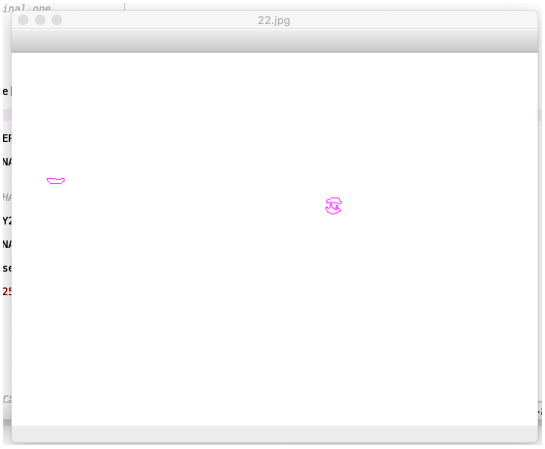

Fig. 10. Case 3, Contours displayed as no bounding box was found

Fig. 11. Case 4

Fig. 12. Case 4, Classification Mask

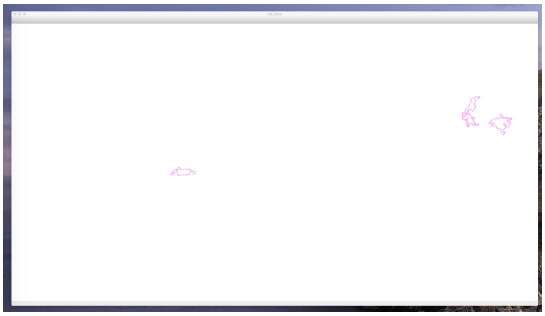

Fig. 13. Case 4, Contours displayed as no bounding box was found

Fig. 14. Case 5

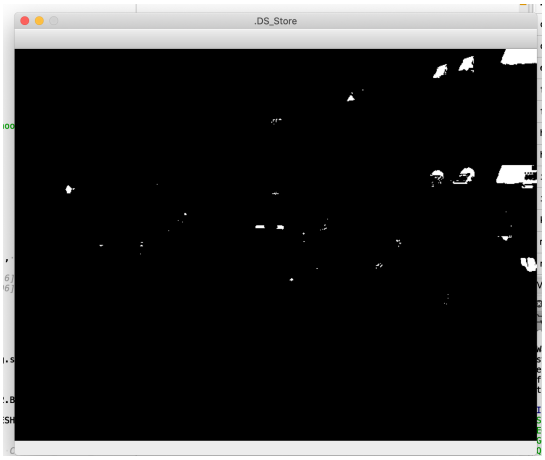

Fig. 15. Case 5, Classification Mask

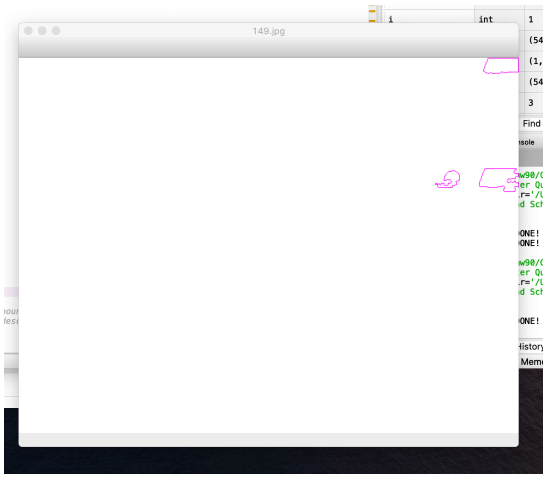

Fig. 16. Case 5, Contours displayed as no bounding box was found

V. Results Discussion

Clearly, for Case 1, the results were as good as one could have hoped for. The segmentation mask displays nearly perfect red and not red, and the bounding box around the stop sign is exactly where it should be, capturing the sign. This case illustrates how the program was able to handle most cases with a sufficiently large stop sign without any overlap with other images. Later cases will demonstrate the cases the program struggled with.

As seen again, the program successfully finds the bounding box of the stop sign and does not mis-identify anything else. It should be noted however that the segmentation image has non-trivial noise, but the shape analysis was able to filter out the non-stop signs.

This is an example of a failed detection. The stop signs are small in the image and it can clearly be seen in the segmentation image that the stop sign was not properly identified, as the stop sign is cut into 2 sections for both stop signs. Since the detection failed, the contours are presented so the reader can see what the shape analysis had to work with when determining if the contours were stop signs. Clearly no good shape analysis would consider these stop signs, therefore this failure is due to the image classifier not being strong enough.

Cases 4 and 5 were both environments where no stop sign was detected, and correctly so. As can be seen in the segmentation masks, and the contours displayed, there were red pixels detected, but the shape analysis was able to determine them not to be stop signs, hence correctly finding no stop signs in the image.

Overall, it can be seen from the results that the program works well overall, and is able to find the stop signs in most cases, and does not mis-idetify any stop signs where there are non, with the exception of small stop signs and low resultion pictures, as seen here.

VI. Discussion with Fellow Students

This student regularly studies in a group with other students Roumen Guha and Maria Fatima and it is acknowledged here that this student discussed this project with these students in capacities such as: the math behind logisitic regression, use of the OpenCV and Numpy libraries in Python (as it is this student's first time using Python), and worked on hand segmenting the training and validation images, as per the professor's allowance.

References

[1] Atanasov, Nikolay. "ECE 276A." UCSD ECE276A: Sensing Estimation in Robotics (Winter 2020), Github, 1 Jan. 2020, natanaso.github.io/ece276a/.