Traffic Sign Recognition

Tudor Cristea
Technical University of Cluj-Napoca
Email: Cristea.Io.Tudor@student.utcluj.ro

Abstract—**This paper presents a method for detection and classification of traffic signs. The approach proposed in this paper highlights the use of various image processing algorithms such as Canny, Hough, MSER and Harris which, if used together, can correctly detect all (or at least, most) traffic signs from an image. The detected outlines are then given to a KNN model that uses, as features, the HOG, LBP and color histogram in order to classify them. The detection part of the application was developed in C++, using the OpenCV library, while the classification part was developed in Python (using several useful libraries, such as numpy, pandas and cv2).**

I. Introduction

In today's rapidly evolving technological landscape, the conversation surrounding *Artificial Intelligence* advancements has captured the attention of people across various domains. One of the central promises of *AI* is to streamline and enhance our daily lives by automating repetitive, mundane tasks. Among these tasks, driving stands out as a prime candidate for automation, given the unexpectedly large amount of challenges it presents, from traffic congestion to driver fatigue. For many individuals, commuting to and from work by car is a necessity, yet it often comes with its own set of frustrations. Traffic congestion, lack of skill or experience, environmental concerns (such as pollution), and encountering rude drivers can all damage the experience of driving. Moreover, there are those who, due to physical disabilities, are unable to safely operate a vehicle, or truck drivers who face the hazards of fatigue during extended journeys. Autonomous driving technology holds the promise of addressing these challenges by providing a safer, more efficient alternative to traditional human-operated vehicles.

One of the key components in the development of autonomous driving systems is traffic sign recognition. Unlike existing solutions that rely on live video feeds from vehicle cameras to process information in real-time, the approach outlined in this paper focuses on image-based processing. This approach involves two main tasks: **detection** and **classification**. The detection task, implemented entirely in the *C++* programming language, takes advantage of the image processing capabilities of the *OpenCV* library. By analyzing input images, the system aims to identify and locate potential traffic signs within the scene. This step is crucial for identifying relevant objects amidst complex backgrounds and varying environmental conditions. Following detection, the classification task, also implemented in *C++* (and possibly *Python*), utilizes a machine learning algorithm to classify the detected traffic signs. This enables the system to learn from labeled data and make predictions about the type of sign present in the input image. The use of a diverse dataset that includes hundreds of images resembling real-world traffic scenarios, is essential for training a robust model capable of handling various lighting conditions (such as shadows or night time photos), occlusions or having multiple distinct traffic signs in the same scene.

II. Related Work

Scouring the Internet, there are many interesting approaches that implement a *Traffic Sign Recognition Application* and it would be fitting to discuss the ideas put forth in some of these papers/articles, before presenting the one proposed in this paper.

The first one involves processing a pre-recorded video, where multiple, different signs are visible and correctly detected [4]. The author affirms that their method follows five main steps:

1. **Segmentation Of The Desired Colors**: Basically, in this step, a histogram equalization is performed, followed by a conversion to the *HSV* color space (which is closer to the way humans perceive colors). Next, the author explains some of the issues that they encountered when identifying the relevant colors (especially red) and the solution they came up with.

2. **Finding The Region Of Interest**: In this step, the author mentions the usage of the *MSER (Maximally Stable Extremal Regions)* algorithm in order to identify certain key areas (or blobs) of interest that match some restrictions. These areas are then plotted on an image which has a black background.

3. **Finding The Contour**: Using the resulted image from the previous step, the author states that they only select three contours which meet a number of reasonable conditions, such as positioning and size. That region is then cropped and resized.

4. **Identifying A Contour**: In this step, the author goes on to explain the method used for training the model on a dataset. However, they stress the fact that, in order to facilitate the identification process, they perform *Feature Detection* before the training phase. In a nutshell, *Feature Detection* is the method of calculating attributes of images which stand out, hence making the identification of similar images easier. Specifically, they use a use a *HOG (Histogram of Oriented Gradients)* feature descriptor which will count the number of times a gradient occurs in a subset of the image. The results are then fed into an *SVM (Support Vector Machine)* training model (which is a supervised one).

5. **Obtaining The Results**: Lastly, the author uses the output probabilities of the model and selects the class which had the highest one. Afterwards, they draw an outlining box around the traffic sign and display the matched image beside it.

Another interesting approach, proposed by four Japanese researchers, introduced the concept of using *Genetic Algorithms* in order to improve the recognition accuracy of traffic signs in adverse conditions such as the sign being shadowed, partially occluded, rotated, twisted or blurred, or even if the image is containing a lot of signs [5]. Moreover, they state that their method does not require a large data set in order to work as well as other approaches that use *Neural Networks (NN)* or *Support Vector Machines (SVM)*, and do require such a set. They also split the application into three stages:

1. **Traffic Sign Detection**: In this stage, the image is first converted from the *RGB* color space to the *HSI* color space in order for the chromatic components to be directly emphasized. Next, the authors have utilized both color features and shape features, which consist of red colour segmentation and circular *Hough* transform.

2. **Traffic Sign Validation**: In this stage, the main objective is to remove the non-plausible candidate regions detected in the previous stage. This is achieved in three processes. The first one involves adjusting the parameters of the *Hough* transform function that is included in the *OpenCV* library in order to obtain less than six possible areas of interest. The second process eliminates the regions that are either too small or too large. The last process extracts the minimum candidate area among overlapping candidate areas.

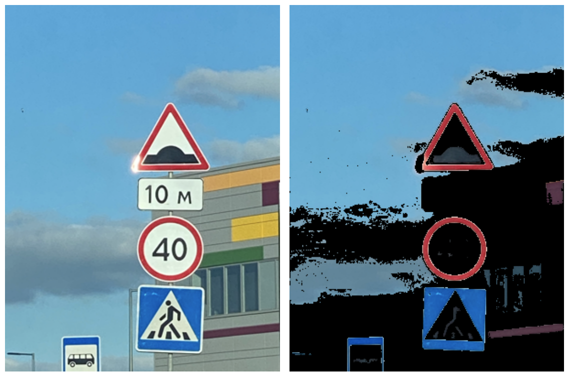
3. **Traffic Sign Recognition**: In this stage, the team uses a number of template images which are modified with the help of gene coding in order to slightly modify the template images, by changing some of their attributes, such as rotation, scale, intensity modulation and smoothness. In this way, not only do they obtain a larger training data set which has slight variations of the original images, but it also facilitates the cross-correlation with images from the validation set. At the same time, this degree of similarity is used as a fitness function, which after multiple evolutions, it must become smaller than a given threshold value.

The last approach that I want to briefly present in this section is heavily dependent on functions which are already implemented in the *OpenCV* library, but the main ideas are still relevant to the task at hand [8]. For starters, the first thing that the author did was to convert the image to the *HSV* color space, similarly to the other two approaches discussed. Next, for stop signs specifically, they extracted the red parts of the image, after which they found the edges and then the contours of the result. Subsequently, they used another function in order to obtain an approximate polygon and if that polygon resembles an octagon, is predominantly red and occupies an area large enough, then it is classified as a ”stop sign”. Similarly, for speed signs, they use a function which implements the *Hough* transform and then they use information about the circle’s center and radius in order to clip out the number part of the image. Finally, they use yet another function which implements the *K-Nearest Neighbor* algorithm and which was prepared beforehand with an image data set containing several hundreds of pictures of numbers.

III. Method

The method proposed in this paper is similar to a processing pipeline consisting of multiple stages through which the initial color image (that can contain one or several traffic signs) passes through and thus, suffers multiple, distinct modifications in order to be able to properly and swiftly detect the areas of interest. Thus, the first task, i.e., **detection** is discussed first.

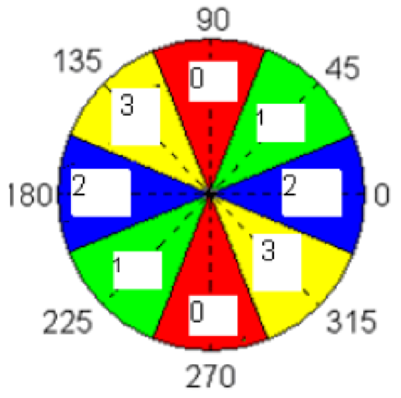
Firstly, a clone of the initial image is converted from the *RGB* (actually, *BGR*) color space to the *HSV* color space. This step helps with representing the colors in a way that is closer to how the human eye perceives color. After this, certain parts of the image which contain some specific colors, are going to be kept, while the rest will be turned to background (i.e., the color black). Since most traffic signs are either blue, or contain at least a red outline (with some exceptions), the resulting image of this processing stage will contain only those parts which could potentially be classified as traffic signs. Therefore, for the blue color, a lower bound and an upper bound have been chosen so that a large enough interval of shades of blue is covered (which takes into account different lighting conditions, orientation and other factors), which is then used to create a mask that will be utilized in order to filter out the unwanted parts and keep only the areas that are ”relevant”. The same procedure has been used for the red color with the addition that, in the *HSV* color space, red can be found in two intervals instead of just one. The two resulting images (one with only red pixels and one with only blue pixels) are then combined in order to obtain an image which contains both of these colors.



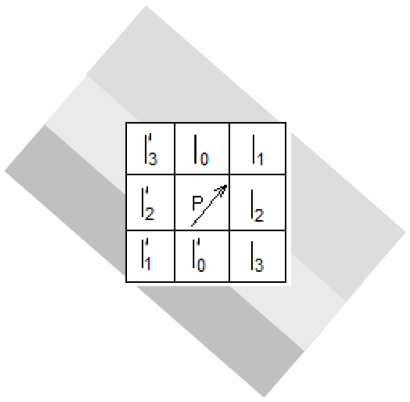
(a) Original image (b) Filtered image

Fig. 1. Filtering Stage

Secondly, after attaining a more narrowed-down version of the initial image, the next process involves finding the edges of the shapes that are present in the filtered image. This is done using the *Canny* algorithm [12] for edge detection. This algorithm consists of multiple steps, the first of which is the conversion of the image to the grayscale color format (which uses different weights for each color channel: 0.299 for the blue color channel, 0.587 for the green color and channel and 0.114 for the red color channel; the reasoning behind this choice for the weights comes from the fact that the human eye has a higher response to the green color than for the other two colors). Next, a (5x5) *Gaussian* filter is applied on the grayscale image in order to obtain a blur effect, thus reducing the eventual noise that might be present in the image. Then, the *Sobel* operators are applied on the image in order to obtain the gradient magnitudes and the directions/angles of the image. By doing so, the edges of the shapes are highlighted. However, they are too thick to be processed in any meaningful way. That's the reason why both results of the last computation are used for a non-maxima suppression algorithm, which essentially, performs a thinning operation on the edges by comparing the magnitudes of three neighboring pixels taken according to the value of the angle of the middle neighbor (this selection process can be seen in figures 2a and 2b), and keeping only the pixels that have a larger magnitude than their corresponding neighbors. However, the goal is to keep only the edge pixels which are the most prominent. For this purpose, an adaptive hysteresis algorithm needs to be employed as well. It uses, at first, two thresholds: a lower one and a upper one, comparing the value of each pixel with these two predefined values and thus, it categorizes the pixel as a ”strong” one if its value exceeds the upper threshold, as a ”weak” one if its value is in between the two thresholds and is outright discarded (turned into a background pixel) if its value is smaller than the lower threshold. The final step of this stage is to perform an algorithm which is very similar to *BFS*, and which will turn the ”weak” pixels into object pixels if they are in the same connected component as (at least) one ”strong” pixel. Otherwise, they are turned into background pixels instead.



(a) Selection of the neighbors for non-maxima suppression



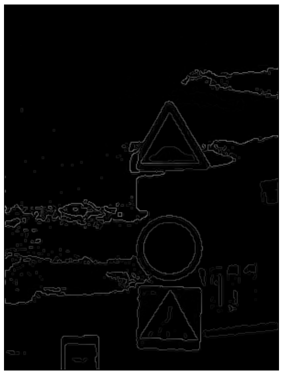
(b) Example for non-maxima suppression

Fig. 2. Non-Maxima Suppression Algorithm

(a) Grayscale image

(b) Gaussian/Blurry image

(c) Sobel-magnitude image


(d) NMS image

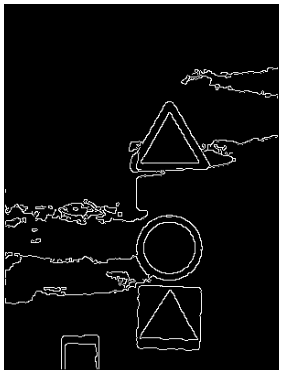

(e) Canny image

Fig. 3. Edge Detection Stage

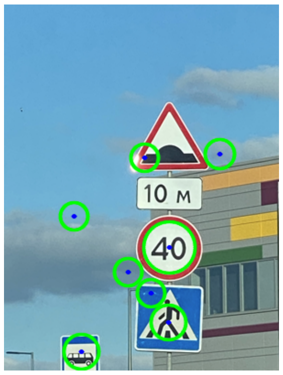
After acquiring an image which essentially contains only the edges of the (most important) objects in the original image, a shape analysis can be performed. This will help with the identification of the traffic signs as well, since some of them have distinct geometrical shapes (for example, the stop sign is the only traffic sign which is an octagon). Mainly, the geometrical shapes which need to be identified are circles, squares, triangles and octagons since these are the most frequent in the data set that is used for this project (but in reality, there are also upside-down triangles, diamonds or rectangles). For circles, an implementation of the *Hough* circles algorithm is used (inspired by the implementation from [6]). The first part of this algorithm consists of building an accumulator that will count the ”votes” for each pixel in the image, using circles of different radii. A ”vote” represents a point in a circle that would be centered at an object pixel and would be located inside the image matrix. Then, a process of selection is performed in order to eliminate the majority of circles which do no constitute suitable candidates for the circles in the image. For this, a threshold is used to filter out the pixels which are less likely to represent circle centers. In order to further reduce the possible number of circle candidates, the circles that were above the aforementioned threshold are sorted in descending order, according to the number of votes and then only the first few tens of them are still kept. In addition, I have introduced a secondary process which further decreases the number of circles by eliminating the circle with less ”votes” from a pair of intersecting circles. A similar implementation was adapted to work with squares as well, by counting the number of ”votes” for a series of squares which are increasing in edge length. However, in order to not introduce an unfair advantage for the squares with larger edge lengths (since they would get more ”votes”), instead of incrementing the accumulator by 1, it is increased by $1/P$, where P is the perimeter of the current square. Unfortunately, it could not be modified further in order to work correctly on triangles, and so, another algorithm, namely the *Harris Corner Detection* algorithm was used for this purpose. However, before applying this algorithm, an important thing to mention is the fact that since all triangular traffic signs have a red outline, this algorithm will try to find triangles using the image which contains only the red color and on which the *Canny* edge algorithm is applied separately. Furthermore, the contours of this image are determined, and then, the *Ramer-Douglas-Peucker* algorithm is applied in order to approximate the shapes of those contours. This way, the number of ”false corners” is reduced significantly. Next, if a point that is both a corner (found by *Harris*) and a point of an approximated contour, then that point is considered to be a ”true corner”. After doing this classification, triangles are found by taking triplets of these ”true corners” and verifying the following conditions:

- one point needs to have a larger y-coordinate than the other two points, while the remaining two points need to have, more or less, the same y-coordinate

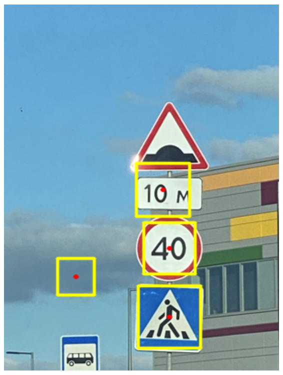
- the lengths of the edges of the supposed triangle need to be inside a certain interval (thus, too small or too large triangles are not taken into consideration)

- the lengths of the edges of the supposed triangle need to be, more or less, equal, since the traffic signs which are triangles resemble equilateral triangles

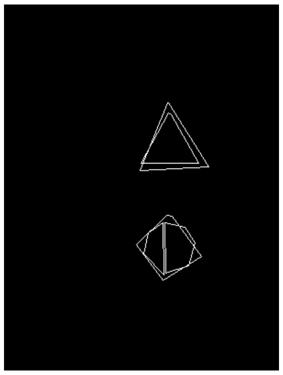
- if one triangle has at least one of its vertices or its center inside another triangle (or vice-versa) and the triangle that is ”inside” the other one has more votes than the encapsulating triangle, then the encapsulated triangle will not be added

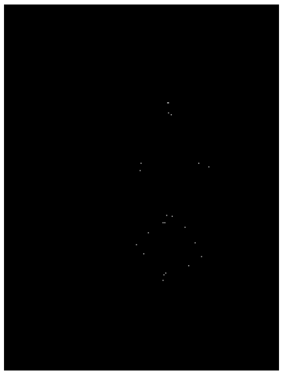


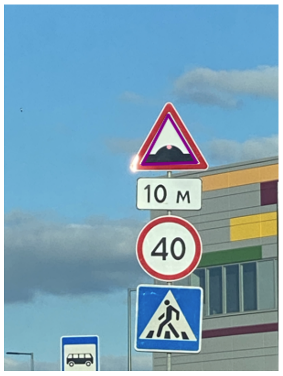
(a) Hough circles



(b) Hough squares


(c) Approximated shapes (red part of the image)

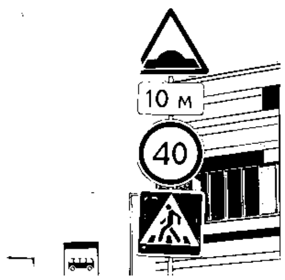

(d) Approximated shapes - points (red part of the image)



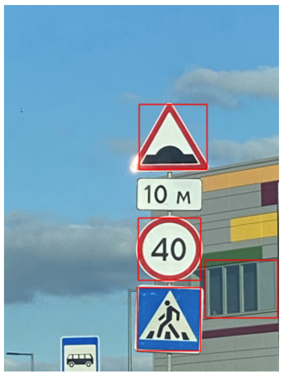
(e) Harris triangles

Fig. 4. Shape Detection Stage - Regular Shapes

In order to ensure that all relevant shapes have been correctly found (including octagons and other shapes which would not be detected by any of the above three algorithms; for example, even a triangle which is slightly twisted to the side would not be detected), the *Maximally Stable Extremal Regions (MSER)* algorithm [3] is employed as well, but this algorithm can only be applied on a thresholded image. For this purpose, *Otsu's* algorithm [2] was used in order to obtain a binary image in which the foreground and the background are fairly separated. In the resulting binary image, the *MSER* algorithm detects "blobs" (i.e., connected components) by using a number of parameters such as, the minimum area and maximum area of a blob, and a threshold which essentially specifies how elongated a blob can be (traffic signs have, in most of the cases, an aspect ratio which is close to 1, i.e., the width is somewhat equal to the height).



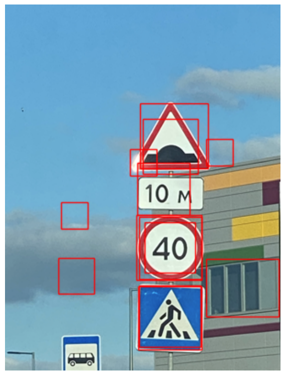
(a) Otsu thresholded image



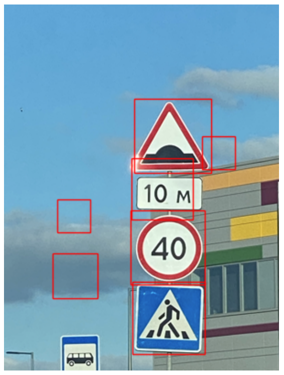
(b) MSER

Fig. 5. Shape Detection Stage - Irregular Shapes

Before classifying the shapes that were found, a filtering mechanism needs to be used in order to reduce their amount, since there are many, and some of them are included inside others. For this reason, for each shape, a corresponding rectangular outline is used. The outlines that are inside another one and for which the area is smaller than that of the bigger outline, are going to be removed. In addition, the outlines that are too close to the edges of the image are also going to be removed since it is highly unlikely that a traffic sign will be close to the edges of the image. Obviously, this method is not appropriate for each image, but it yields very good results for most of them, since we want to at least detect correctly all (or at least, most) traffic signs in every picture even if some "false positives" are caught in the process as well.



(a) Semifinal image



(b) Final image

Fig. 6. Final Stage

Now that the shapes of the traffic signs have been successfully detected (albeit containing some impostors in some of the cases), a separate image is created for each outline separately.

The second (and actually, last) task, i.e. **classification** is discussed next. In order to facilitate this process, a *Machine Learning* algorithm is used, namely the *K Nearest Neighbor*

(KNN) algorithm [11]. This algorithm uses the K closest neighbours from the training dataset in order to classify an example from the test set. It is doing so by computing (in this case), the *Euclidean* distance between the features of the current test example and the corresponding features of all the examples from the training set (in order to make it run faster and use less memory, broadcasting was used; this means that the terms of the computation, i.e., arrays, are expanded if needed, and loops are avoided). In addition to the usual *predict()* method, an implementation of the *predict_proba()* method is employed as well in order to eliminate the aforementioned impostors that were introduced by the detection stage of the application, by looking at the probabilities of the predictions for each class, and taking the result of the prediction into consideration if and only if the maximum computed probability is above a certain threshold (0.85, in this case). The dataset that was used for training, validation and initial testing is the infamous *German Traffic Sign Recognition Benchmark (GTSRB)* [10], from which the classes of signs that do not appear in our test data set, were removed in order to not train the model on data that is irrelevant to us and to not waste memory for no reason.

The set of features that is used for this algorithm is the following:

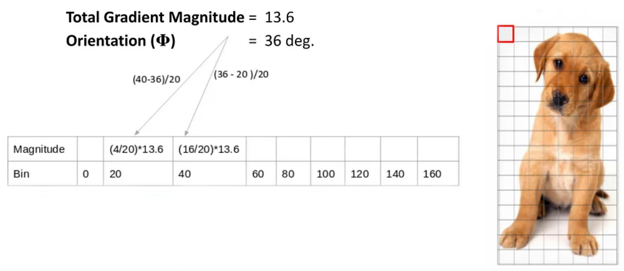
- the concatenation of three histograms, one for each color channel (red, green and blue)

- the histogram of oriented gradients (*HOG*) of the corresponding grayscale image [9]

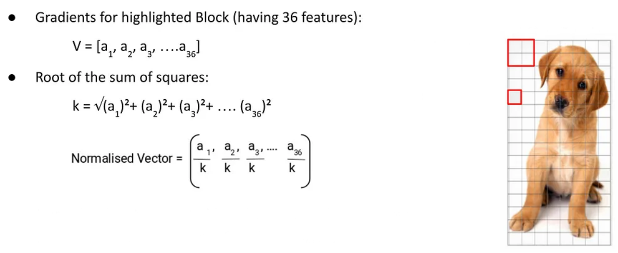
- the local binary pattern (*LBP*) histogram of the corresponding grayscale image [1] [7] (works particularly well together with *HOG*)

In order to ensure that all images have the same number of elements for each of these features, they are resized to the same dimensions, namely to 64x128.

As the name suggests, for the *HOG*, the magnitude and the orientation of a grayscale image (similar to Canny) are computed first, but not by using the *Sobel* kernels for computing the x and y gradients, but instead by using the following formulas: $G_x(r,c) = I(r,c+1) - I(r,c-1)$ and $G_y(r,c) = I(r-1,c) - I(r+1,c)$. Then, the magnitude is split into 8x8 cells, and for each of these cells, a corresponding histogram with 9 bins is computed. This is done by looking at the value of the orientation for a certain pixel and identifying the two bin values in between which this would need to be placed (the bin values, in degrees, are: 0, 20, 40, 60, 80, 100, 140 and 160). For those two identified neighbouring bins, fractions of the magnitude are added to them according to how close the orientation is to each of them. The last step is to normalize the obtained histograms by grouping them into groups of 4, and computing the normalization factor so that every value from those four histograms is divided by it.



(a) HOG - histogram computation example


Gradients for highlighted Block (having 36 features):
V = [a1, a2, a3, ...a36]
Root of the sum of squares:
k = √(a1)²+ (a2)²+ (a3)²+ .... (a36)²
Normalised Vector = (a1/k, a2/k, a3/k, ... a36/k)

(b) HOG - histogram normalization example

Fig. 7. HOG

The first step in computing the *LBP* histogram is to divide the image into 16x16 cells. Then, each of these cells needs to be traversed, and for every pixel in a cell, look in its 8-neighborhood and use 8 values (one for each neighbor) which can be either 0 if the center value is larger than that neighbor or 1, otherwise. By choosing a certain ordering (such as starting from one of the neighbors and traversing them clockwise or counter-clockwise), an 8-bit binary number is obtained. Next, this number is converted to decimal and a histogram with 256 bins is computed based on these decimal values, for each cell. Optionally, these histograms can be normalized before returning the final result.

IV. Evaluation and Results

The metrics that were chosen in order to show the performance of the model are very common in classification problems, namely the accuracy, precision, recall and f1 score. The values that were obtained for these four metrics on the default test set are shown in the table below. For the number of neighbors, a value of 17 yielded the best results.

Metric\Class	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 7
Accuracy	0.791						
Precision	0.516	0.573	0.684	0.577	0.895	0.765	0.675
Recall	0.939	0.834	0.626	0.673	0.801	0.523	0.734
F1 Score	0.666	0.679	0.653	0.622	0.846	0.621	0.703

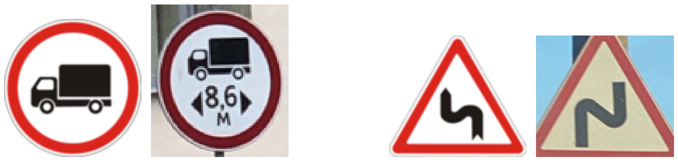
Metric\Class	Class 8	Class 9	Class 11	Class 12	Class 13	Class 14	Class 15
Accuracy	0.791						
Precision	0.797	0.9	0.8	1.0	0.995	0.933	0.857
Recall	0.699	0.915	0.785	0.989	1.0	1.0	0.967
F1 Score	0.745	0.907	0.792	0.994	0.997	0.965	0.909

Metric\Class	Class 16	Class 17	Class 21	Class 22	Class 26	Class 27
Accuracy	0.791					
Precision	0.966	0.891	0.766	0.666	0.666	0.783
Recall	0.966	1.0	0.663	0.555	0.530	0.783
F1 Score	0.966	0.942	0.711	0.606	0.591	0.783

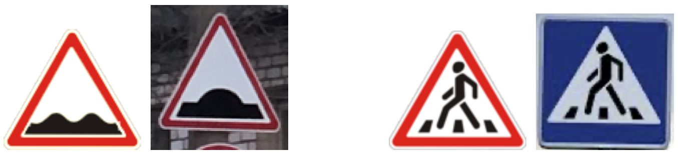
Metric\Class	Class 28	Class 29	Class 30	Class 35	Class 36	Class 37
Accuracy	0.791					
Precision	0.546	0.677	0.346	0.933	0.833	0.416
Recall	0.589	0.924	0.547	0.957	1.0	1.0
F1 Score	0.567	0.782	0.424	0.945	0.909	0.588

TABLE I
TEST SET PERFORMANCE METRIC RESULTS

However, the results that were obtained for the test set that was generated during the detection stage of the application were not as good and this is due to the discrepancy in the two datasets since some signs from the training dataset do not appear in the test dataset and vice-versa, or two signs are slightly different which is enough to not properly classify it (a few examples are shown in the image below, where, for each pair, on the left, are the signs that are in the *GTSRB* dataset while on the right are the corresponding signs that are detected from my dataset). Moreover, the images on which the detection was performed usually contained multiple different signs, and thus the problem became a multilabel classification one, further negatively affecting the performance of the model. For the precision, recall and f1 scores that are presented below, the average of all classes was computed.



(a) (b) (c) (d)



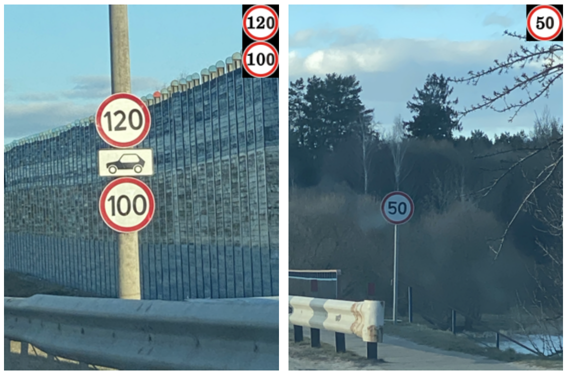
(e) (f) (g) (h)

Fig. 8. Similar signs

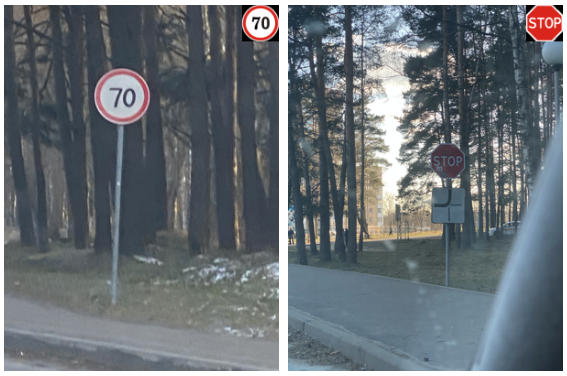
Metric	Value
Accuracy	0.127
Precision	0.251
Recall	0.099
F1 Score	0.083

TABLE II
PERSONAL TEST SET PERFORMANCE METRIC RESULTS

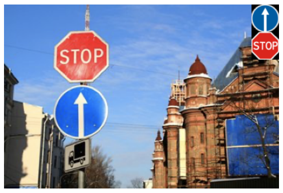
Even though these results are not pretty to look at, there were some cases in which the model did a very good job at correctly classifying multiple signs from an image (even in some harsh lighting conditions). Some of the best results are shown below.



(a) (b)



(c) (d)



(e)

Fig. 9. Best results

V. CONCLUSION

In this paper, a unique solution to the traffic sign recognition problem was presented. The proposed system consists of two stages: detection and recognition. In the detection stage, only the relevant colors were filtered (namely, red and blue), after which the appropriate shapes (circles, squares and triangles) were detected by multiple collaborating algorithms. In the recognition stage, a *KNN* model was trained and then tested on the proposed dataset, and while the initial results were promising, the various discrepancies between the two different datasets ultimately could not be avoided, and thus, the final results were a bit disappointing. However, I am confident that if matching datasets were used for both training and testing, the obtained performance of the proposed approach would have been much better.

References

[1] Local binary patterns. *Wikipedia*.

[2] Understanding otsu's method for image segmentation. *Baeldung*, 2023.

[3] S. Bellary. Mser (maximally stable extremal regions). *Medium*, 2023.

[4] S. Gupta. Traffic sign detection & recognition. *Medium*, Life and Tech, 2019.

[5] M. Kobayashi, M. Baba, K. Ohtani, and L. Li. A method for traffic sign detection and recognition based on genetic algorithm. *2015 IEEE/SICE International Symposium on System Integration (SII)*, pages 455–460, 2015.

[6] C. Lemus. A circle hough transform implementation using high-level synthesis. pages 11–19, 2020.

[7] M. Pietikäinen. Local binary patterns. *Builtin*, 2010.

[8] P. Thrane. Opencv and sign recognition. *Wordpress*, 2016.

[9] M. Tyagi. Histogram of oriented gradients: An overview. *Builtin*, 2024.

[10] H. Vagadia. German traffic sign recognition benchmark. *Medium*, 2020.

[11] R. Varga. Pattern recognition systems – lab 8: K-nearest neighbor classifier.

[12] A. Williams. Image processing algorithms: Canny edge detector. *Medium*, 2021.