ASSIGNMENT 2 REPORT

Irish Speed Signs Detection and Classification

RE4017 - Dr Colin Flanagan

Finn Hourigan 17228522

Ronan Reilly 18242367

Brendan Lynch 18227651

Barry Hickey 18243649

Contents

Introduction

In light of the growth of autonomous transportation and the development of intelligent transport systems (ITS) to augment road safety, traffic management and pollution from road transportation, a crucial aspect of such a system is the ability to detect road signs in front of the vehicle. Speed signs in particular need to be differentiated from other sign types on the road as they are fundamental in maintaining safe driving speeds on particular roads. Individually detecting these signs would prove time consuming and there is a high chance of error occurring, particularly in places that more than one sign may be visible to the camera. In trying to address this issue, developing a speed sign detector programme can enhance the effectiveness and efficiency of detecting speed limit signs that are inputted in as regular RBGA images.

This task involves accurately detecting speed signs regardless of their location in the given image or the lighting in the given image. The program is required to distinguish between speed limit signs and other types of signs that can be found on roadsides, such as directional signs, STOP signs and warning signs.

The primary objective of this project is to develop a robust and efficient speed limit sign detector programme that can effectively point out speed sign limits in real-time, regardless of the lighting or sign position. The program needs to be able to remove/ignore objects from the image that are not of interest to detecting the speed limit, including objects that have similar colours to that found in the speed signs.

Figure 1 - Example of sign to be detected

Operation of detector

1) Open Anaconda prompt
2) Navigate to location of file signdetector.py
3) Ensure that all images to be tested and files roifinder.py and classifier.py are all in the same directory.
4) Enter command into Anaconda prompt: python singdetector.py imgfile.png
   , where imgfile.png specifies the name of the image.

   See example command:

```
C:\RE4017\Proj2>python signdetector.py 120-0005x1.png
```

5) Project will run and return the value of the speed limits on detected signs via the terminal and will display the original image overlayed with a text label slightly above the detected sign.

Methodology

As previously highlighted, the requirement of this project is 'Given an image of a road scene, detect *all instances* of speed signs in it (40, 50, 60, 80, 100, 120km/h signs only)'. The detections must also be reported in the terminal window of the console. The programme must not detect any signs that are outside the aforementioned set.

The architecture used to create this program is a proposer/classifier architecture. This is an object detection framework used in computer vision that allows for the detection and classification of objects in an image It works in the following two stages:

1. Region Proposer:
   The first stage of this architecture is described as region proposal. In this stage of the architecture, Regions of Interest (RoIs) in the inputted image are created. These regions are the areas containing the road signs. The main objective in outlining these regions is to reduce the amount of data that the classifier needs to work through into a manageable data set.
   The region proposer in this project is developed based on the HSV colours in the given image. It examines the inputted image and returns the areas that are most likely to contain road signs (RoIs). There can be some mistakes made by the region proposer but they can be filtered out at later stages. This method of generating regions of Interest is quite quick.

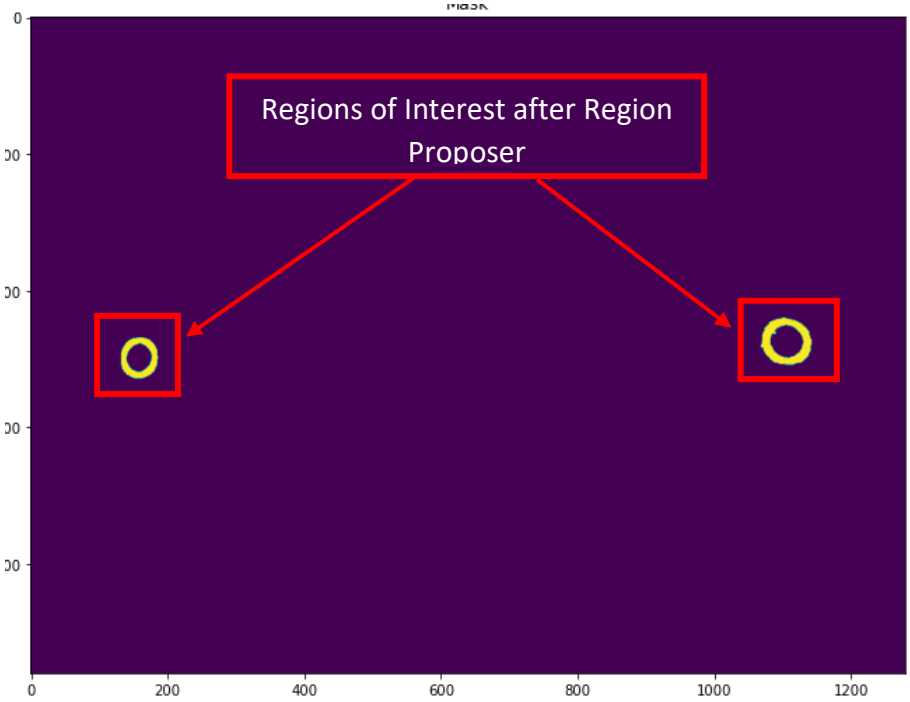


Figure 2 - ROIs highlighted in the Region Proposer

2. Region of Interest Classifier:

   The RoI classifier stage or module of the architecture examines each given region to see if it contains a road sign. This element of the architecture can be the slower aspect of the whole programme but as the regions inputted to it are much smaller, it should be quite accurate. The classifier is used to arrange the images into different categories based on their visual appearance, in this case using a k-NN (*'k – Nearest Neighbour')* network.

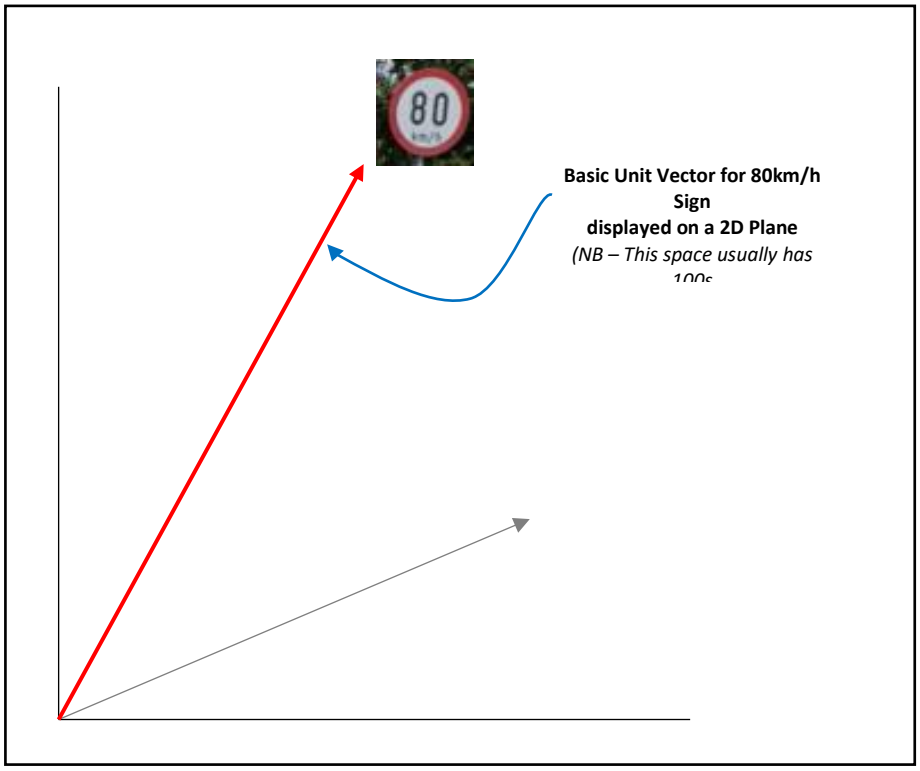


Figure 3 - kNN Classifier Model

The suggested program architecture as below was used:

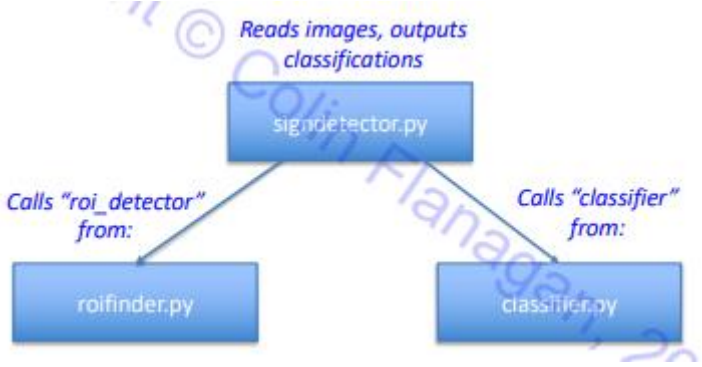


Figure 4 - Program Architecture (Colin Flanagan, 2023)

Signdetector.py calls suitable functions from rpofinder.py and classifier.py, and uses them to generate the required output, the original image with a text box overlayed indicating the value of speed sign(s) present. Each file's contents are discussed in detail in this report.

Region Proposer (roifinder.py):

The function which is called by signdetector.py from roifinder.py is called findROIs (img_name). This takes an image file name as an argument and returns an array of regions of interest as numpy arrays of RGB images, as well as the top right-hand corner co-ordinates of these ROIs.

Image Format Conversion

The images that are to be inputted to the program are in the png format. They are in the RGBA colour space. This refers to Red, Green, Blue and Alpha (greyscale) channels in the colour space. Each pixel is described using four numbers, one for each of the four channels in their respective order. In this colour space, the images cannot be processed. In order for them to be processed, they need to be brough to the HSV space. HSV colour is determined by the Hue, Saturation and Value of the pixel.

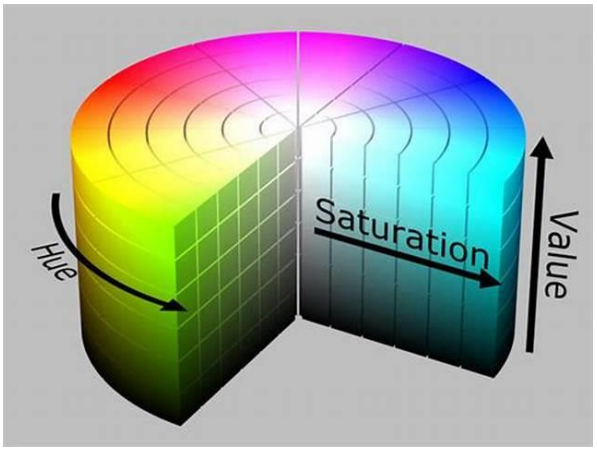


Figure 5 - HSV Colour Scale

Hue refers to the position of the colour on a spectrum. It is given by the angle between 0-360 degrees . Saturation refers to the intensity of the colour from weak t to strong and value refers to the brightness of the colour from dark to bright.

In order to transfer an image from the RGBA colour space to the HSV colour space, it must first be converted to RGB. The skimage library has functions to convert the RGBA image to RGB and a further function to convert from RGB to HSV. Once the image is in the HSV colour space it can be worked with.

```
#Convert from RGBA to RGB
img_rgb = rgba2rgb(img_rgba)

#Convert from RGB to HSV
img_hsv = rgb2hsv(img_rgb)
```

Figure 6 - RGBA to HSV conversion code

In order to threshold the three channels of the HSV image, they must me separated into their respective channels. The Hue figure is in the first column, the saturation figure is in the second column and the value figure is in the third column (first column in the array is column 0).

```
# Split HSV into single channels

img_h = img_hsv[:,:,0]
img_s = img_hsv[:,:,1]
img_v = img_hsv[:,:,2]
```

Figure 7 - HSV image split into 3 channels (Hue, Saturation and Value)

Thresholding the Image

The method of identifying the speed limit signs from the images is by their red rim surround. All speed signs contain this rim and it makes it generally easily interpreted from the rest of the image.

```
# Choose threshold values for H,S and V in format [lower_limit , upper_limit]
# Any pixel WITHIN threshold limits -> 0
# Any pixel outside threshold limits -> 1

th_h = [0.025, 0.675]
th_s = [0, 0.3]
th_v = [0, 0.22]
```

Figure 8 - Thresholds for H, S and V channels

The upper and lower limits can be seen for each of the hue, saturation and value channels respectively. These limits were created through analysis of the HSV colour spectrum and trial and error in watching how the thresholds worked with the set of images provided, and what values were effective at identifying red speed sign rims.

Building the image mask

In order to build a single mask for the inputted image, each channel bust be masked individually and then combined. To create the mask for each channel, a mask helper function was defined, in which each pixel is binarized depending on whether they lie within the threshold limits. If they do, they are given a value of 0, otherwise they are given a value of 1. The outputted data of this function is a binarized image array.

```
# define mask function , with np array of image and thresholds as parameters

def mask(img_arr, th):

    # Determine rows and columns of array
    r,c = img_arr.shape

    # Determine upper and lower threshold limits
    th_up = th[1]
    th_lo = th[0]

    #print(img_arr[140,1000])

    for i in list(range(0,r)):    #loop through rows

        for j in list(range(0,c)):    #loop through columns

            #binarize to 1 or 0 depending on threshold limits
            if img_arr[i,j] <= th_up and img_arr[i,j] >= th_lo :
                img_arr[i,j]= 0 #Any pixel within threshold limits -> 0
            else:
                img_arr[i,j] = 1 #Any pixel outside threshold limits -> 1

    #print(img_arr[140,1000])
    return img_arr
# Use mask function defined above to define threshold

img_h = mask(img_h,th_h)
img_s = mask(img_s,th_s)
img_v = mask(img_v,th_v)
```

Figure 9 - Mask Function Code

The masked images from each channel ***(img_h, img_s, img_v)*** are then combined. They are multiplied together in order to create a single masked image of the three channels. This method is effective in the manner that if any of the three channels holds a given pixel as 0, the final pixel will be turned into a zero due to the simple mathematical principle of multiplying anything by zero = zero. This in turn only keeps pixels in the combined mask that are within the respective threshold in all three of the channels.

```
# Combine binarized channels into 1 mask
img_mask = img_h * img_s * img_v
```

Figure 10 - Combining the single channel masks

Refining the Image Mask

This mask alone leaves noise in the image as there are some regions in the image that are within the thresholds that are not speed limit signs. In order to remove some of these the masked images are filtered using the maximum filter initially. This increase the areas binarized as 1 by thickening the binary-true pixels by 5x5 shape and thus connects nearby regions of sign borders to form one region. Following this maximising, a binary erosion filter is applied to the image in the form of a NumPy array in a plus shape.

```
#Max filter applied
img_mask = scipy.ndimage.maximum_filter(img_mask,5)

# #Binary erosion filter
plus_shape= np.array([[0,1,0],[1,1,1],[0,1,0]])
# #print(plus_shape)

img_mask = scipy.ndimage.binary_erosion(img_mask,plus_shape)
```

Figure 11 - Filtering Regions in Region Proposer

Label binary regions

Binary regions then need to be labelled. These then are given an integer from 1 to max number of connected regions in the image.

```
#Use ndimage.label to generate a labeled array and find the number of labels used
labeled_array, num_features = ndimage.label(img_mask)
```

Figure 12 - Uniquely labelling each joined region in the mask

The **ndimage.label** function is used to generate a labelled array where each connected component in the binary image is assigned a unique integer label. The second returned value**, num_features**, is the integer number of unique labels (i.e., the number of connected components).

Find size (no. of pixels) of each region

Next, the code computes the size of each region in the labeled array by iterating over the range of unique labels (1 to **num_features**) and using the **np.count_nonzero** function to count the number of non-zero pixels in the corresponding region mask. The region sizes are stored in the region_sizes list.

```
# Get the size of each region and store in an array
region_sizes = []
for i in range(1, num_features+1):
    region_mask = labeled_array == i #region_mask refers to only areas where value is i
    region_size = np.count_nonzero(region_mask)
    region_sizes.append(region_size)
```

Figure 13 - Finding the number of pixels in each region

Finding the size threshold

The largest region is then identified by finding the index in region_sizes with the maximum value using **np.argmax.**

```
# Find the index of the largest region
largest_region_idx = np.argmax(region_sizes)

# Compute the size threshold for removing small regions
size_threshold = 0.3 * region_sizes[largest_region_idx]
```

Figure 14 - Size Thresholding Code

A size threshold is then computed as 30% of the size of the largest region. The logic here is that the sign is likely to be significantly larger than any other small and incorrectly picked up binary regions, and in the case of images with 2 speed signs that the smaller region isn't less than 30% of the size of the larger sign region.

Define a minimum ROI size

```
min_size = 576
```

Figure 15 - Minimum size of accepted ROI

This value may be needed in the case where no speed sign is present, and therefore the ROIs determined after masking are small but won't be removed by the threshold as the largest region is in fact one of these small regions! . A 24 x 24 image (576 pixels) was decided as appropriately small for a ROI to not in fact be a discernible speed sign and more likely to just be a small region which contained pixels within the given HSV thresholds declared.

Using size thresholds to remove small regions

The code iterates over the labeled regions again and stores the label values of ROIs whose size is greater than or equal to the size threshold in the **labels_to_keep** list. All small regions below the threshold values (**size_threshold** and **min_size**) are removed by setting their pixel values in the **labeled_array** to zero.

```
# Remove small regions and store labels of large regions (our ROIs)
labels_to_keep = [] #To store label values of ROIs

for i in range(1, num_features+1):
    if region_sizes[i-1] < size_threshold or region_sizes[i-1] < min_size:
        # Small regions below the threshold value or min size value are removed (set to 0)
        labeled_array[labeled_array == i] = 0
    else:
        # Any regions not removed have their label values added to labels_to_keep array
        labels_to_keep.append(i)
```

Figure 16 - Removal of small regions based on previous threshold

The **labels_to_keep** list can then be used to extract the corresponding regions of interest from the original image.

Bounding Box helper function

The helper function **find_bounding_box** takes a **label** (integer) and a labelled array containing that label, and outputs a bounding box.

```python
def find_bounding_box (label, labeled_array):
    region_mask = labeled_array == label
    indices = np.nonzero(region_mask)

    min_x, max_x = np.min(indices[1]), np.max(indices[1])
    min_y, max_y = np.min(indices[0]), np.max(indices[0])
    bbox = (min_x, min_y, max_x, max_y)

    return bbox
```

Figure 17 - Bounding box function

Region_mask refers to an array the same size, but only including the region with the argument label. The **indices** variable contains an array of co-ordinate pairs (N x 2 array) of each non-zero pixel (labelled pixel) in the **region_mask** array. Using the **np.min** and **np.max** methods, we can extract from this array the values for **min_x**, **min_y**, **max_x** and **max_y** of the labelled region. These values sufficiently describe the co-ordinates of the bounding box of the region , as below:

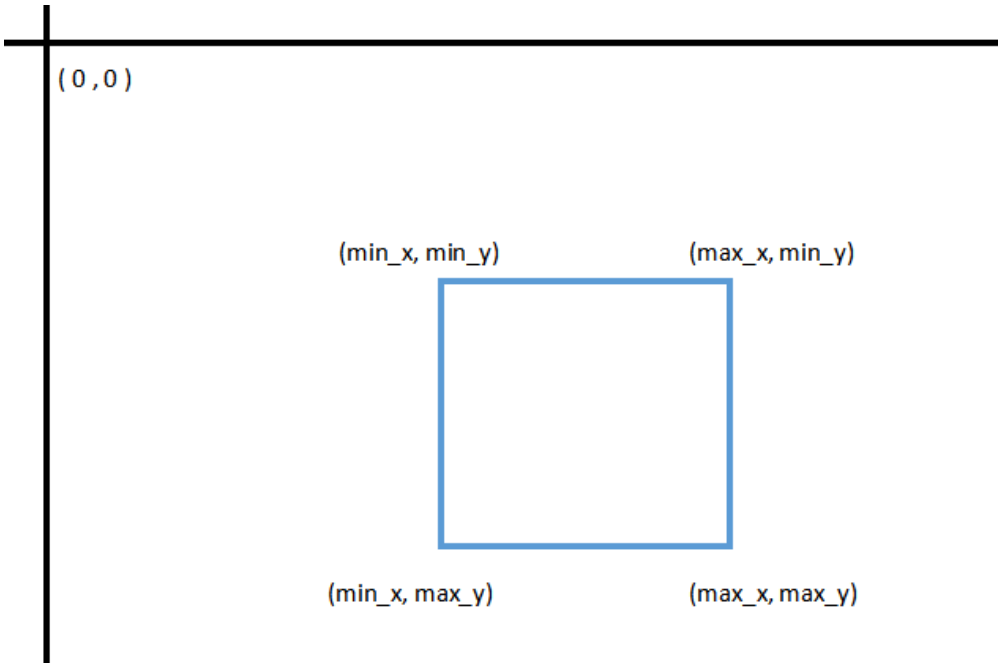


Figure 18 - Definition of Bounding Box Co-Ordinates

These values are returned in an array named **bbox** in the format **[min_x, min_y, max_x, max_y]**

Finding bounding boxes / slicing ROIs from RGB array / removing narrow regions

Our code is iterating through a list of labels called **labels_to_keep**, and for each label in this list, it finds the bounding box coordinates of the corresponding region in a labelled array using the **find_bounding_box()** helper function. Once it has the bounding box coordinates, it calculates the length of the horizontal and vertical sides of the bounding box, as well as the co-ordinates of the top

right-hand corner of the box.

```
# Find the bounding box of each region using find_bounding_box function
for label in (labels_to_keep):
    bbox = find_bounding_box (label, labeled_array)

    #Extract values from bbox array
    (min_x, min_y, max_x, max_y) = bbox
    #Length of horizontal side of box
    deltax = max_x - min_x
    #Length of vertical side of box
    deltay = max_y - min_y
    #Co-ordinates of top RH corner of box, needed for adding text to image at later stage
    top_corner = (max_x,min_y)

    #Remove narrow regions with a ratio greater than minratio for small side/bigger side
    minratio = 0.6

    # If both the ratio of height/width and width/height are above the minimum ratio,
    # the region_array is added to the array of ROI arrays
    if (deltax/deltay > minratio and deltay/deltax > minratio ):
        #Slice the RGB image to extract the subarray corresponding to the bounding box
        region_array = img_rgb[min_y:(max_y+1), min_x:(max_x+1)]
        print(f"Bounding box of region {label}: {bbox}")
        # Add ROI to storage array
        roi_array.append(region_array)
        # Add top corner co-ordinates to array (at corresponding index to ROI)
        top_corner_array.append(top_corner)
```

Figure 19 - Code to Check aspect ratio of the bounding box

The code then checks whether the ratio of the height to width and the ratio of the width to height are both greater than a minimum ratio **minratio**. If both ratios are greater than **minratio**, it slices the RGB image to extract the subarray corresponding to the bounding box, adds it to an array of region of interest (ROI) arrays called **roi_array**, and adds the co-ordinates of the top right-hand corner of the bounding box to an array called **top_corner_array**.

Finally, it prints the bounding box coordinates of the region, so that the user can check that the correct regions are being extracted.

Return Values

```
print("ROI search complete...")

return roi_array, top_corner_array
```

Figure 20 - Values returned from ROI Classifier

Finally, a message prints to the terminal indicating that the ROI search is complete. The return values of resulting ROI arrays and top corner co-ordinates can be used for further processing.

Region of Interest Classifier:

The function **classify_img()** is called from signdetector.py. This function is for classifying a region of interest (roi) as a given speed sign value. It determines the speed sign value according to our 1-NN classifier and the given descriptor vectors file (1-NN-descriptor-vects.npy).

Takes a numpy array representing RGB data of the region of interest as an argument.

Returns the speed sign value determined.

Creating a grayscale, 64 x 64, contrast-enhance version of argument ROI

Initially, the code is converting the given colour image to grayscale using the **rgb2gray()** function. Then, the grayscale image is resized to a 64x64-pixel image using the **resize()** function with anti-aliasing.

```
# convert to grayscale
gray_img = rgb2gray(roi)


# Resize the image to 64x64 pixels

resized_img = resize(gray_img, (64, 64), anti_aliasing=True)

#Determine max and min grayscale values of array
max_g = resized_img.max()
min_g = resized_img.min()

#Rescale to int values 0 - 255 and contrast enhance by using full 0 - 255 scale
resized_img = (resized_img - min_g) * (255/(max_g - min_g))
```

Figure 21 - Grey scaling Image and contrast enhancement

Next, the maximum and minimum grayscale values of the resized image array are determined using the NumPy **max()** and **min()** functions. Finally, the grayscale values of the image array are rescaled to the range of 0 to 255 for better visualization and contrast enhancement. This is achieved by subtracting the minimum grayscale value and multiplying by a factor that scales the values between 0 to 255.

Creation of descriptor vector for the given ROI

First, the mean pixel value of the resized image is calculated using the NumPy **mean()** function. This value is then subtracted from all the elements of the resized image to obtain the "zero mean image".

```
# Find mean pixel value of resized image
mean_pixel_value = np.mean(resized_img)

# Subtract the mean pixel value from all elements of resized image to find the "zero mean image"
zero_mean_img = resized_img - mean_pixel_value

# Create a 4096-elemnt vector from the 64 x 64 "zero mean image" using .fLatten()
img_vector = zero_mean_img.flatten()

# Find the value of the norm for this vector
norm =np.linalg.norm(img_vector)

# Normalise vector by dividing by the norm value
# This 4096 element vector is to be compared to the descriptor vectors supplied
norm_img_vector = img_vector/norm
```

Figure 22 - Creation of our ROI's own Descriptor Vector

Next, the **flatten()** method is used to create a 4096-element vector from the 64 x 64 "zero mean image". The resulting vector is then normalized by dividing it by its Euclidean norm, which is calculated using the **linalg.norm()** function.

The resulting normalized 4096-element vector is then used as a descriptor vector that can be compared to other descriptor vectors as part of the 1-NN classification method.

Comparison of generated descriptor vectors and library of descriptor vectors

Loading descriptor vectors from an **.npy** file is performed using NumPy's **load()** function. The first column of the matrix corresponds to categories (speed sign values), while the remaining columns correspond to descriptor vectors. The code assigns these values to **categories** and **template_vector_set** respectively.

```
#Load descriptor vectors .npy file
mat = np.load("1-NN-descriptor-vects.npy")

#Categories are the first column (index 0)
categories = mat[:,0]

#Descriptor vectors are the remaining columns
template_vector_set = mat[:,1:]
```

Figure 23 – Creating arrays called categories and template_vector_set

The code initializes an empty array **distances** for storing the distance values between two vectors. It then iterates through all the categories by looping through **len(categories)** using the for loop.

```
#Array for storing the distances between vectors
distances = []

for i in list(range(len(categories))):
    #Template vector from file defined as all values in row i, a 4096 element vector
    template_vector = template_vector_set[i,:]
    #Compare template_vector & norm_img_vector using function 'distance_between_vectors' function
    dist = distance_between_vectors (template_vector, norm_img_vector)
    #Add the distance between them to storage array 'distances'
    distances.append(dist)


# Find the index of the smallest values in array 'distances'
# This is our 1-NN classifier's closest match
smallest_dist_idx = np.argmin(distances)
speed_limit = categories[smallest_dist_idx]
```

Figure 24 – Find distance between ROI vector and sample vectors, determine closest match

Within the loop, it gets the template vector by selecting a row from **template_vector_set** and compares it with the normalized image vector using a custom function called **distance_between_vectors()**.

```
def distance_between_vectors (vector1, vector2):

    # finding sum of squares
    sum_sq = np.sum(np.square(vector1 - vector2))

    # Doing squareroot
    dist = np.sqrt(sum_sq)

    return dist
```

Figure 25 - Function for comparing Distances between generated vector and template vectors

The resulting distance value is then appended to the **distances** array.

After computing the distance values for all the categories, the code uses NumPy's **argmin()** function to find the index of the smallest distance value in the **distances** array. This index corresponds to the

category that has the closest match with the normalized image vector generated from the input ROI array. Finally, the category is assigned to the variable **speed_limit**.

Return Value

```
print("Classifier complete...")
return speed_limit
```

Figure 26 - Returned Classifier Value Code

A message indicating that the classifier function has finished is printed to the terminal. The value for **speed_limit** is the return value.

Sign detector (signdetector.py)

This file is the python file which is run in order to process an image file and detect any speed signs are present.

Command line argument

The image file's name (including extension) is entered as a command line argument. To use this string to load the given image at any stage, it must be allocated to variable **image_name** using the following sys command:

```
img_name = sys.argv[1]
```

Figure 27 - Extracting command line argument as string

Finding ROIs

Our code calls the function **roifinder.find_ROIs()** from roifinder.py to detect regions of interest (ROIs) in the image that potentially contain speed limit signs. The **find_ROIs()** function returns two arrays, one containing the detected ROIs, and the other containing the top-right corner coordinates of each ROI.

```
#Find all ROIs and their corresponding top RH corner co-ordinates
roi_array, top_corner_array = roifinder.find_ROIs(img_name)


# Speed limit results storage array initialised
speed_limits_arr = []

#Find speed limit associated with each roi using classifier, add to storage array
for roi in roi_array:
    speed_limit = classifier.classify_img(roi)
    speed_limits_arr.append(speed_limit)
```

Figure 28 - Using roifinder.find_ROIs() and classifier.classify()

Next, the code initializes an empty array called **speed_limits_arr** to store the detected speed limit values for each ROI. A for-loop iterates through each ROI in the **roi_array** and uses the **classifier.classify_img()** function from classifier.py to classify the image entered as the argument and return the speed limit value associated with that ROI. The speed limit value is then appended to the **speed_limits_arr** array.

Define locations of text label and text background rectangle

Translations of the co-ordinates in **top_corner_array** are required to define the location of where the text for a detected sign should be placed. Furthermore, the corners of the background rectangle are also specified relative to the text location.

```
# Translation operation, co-ordinates of text location defined
text_loc = [(x - 100, y - 60) for x, y in top_corner_array]

# Translation operation, co-ordinates of top LH corner of label box defined
box_corner1 = [(x - 20, y - 20) for x, y in text_loc]
# Translation operation, co-ordinates of bottom RH corner of label box defined
box_corner2 = [(x + 230 , y + 80) for x, y in box_corner1 ]
```

Figure 29 - Required co-ordinate translations

Addition of sign text label(s) to original image

The code loads an image specified by the variable **img_name**, creates a new image with the same dimensions as the original image and fills it with a white color. The original image is then copied onto the new image using the **paste** method.

```
# Load the original image
img = Image.open(img_name)
width, height = img.size

# Create a new image (blank, same dimensions)
img_with_text = Image.new('RGB', (width, height), color=(255, 255, 255))

# Copy original image onto the img_with_text
img_with_text.paste(img, (0, 0))
```

Figure 30 - Creation of copy of original image

An ImageDraw object is created on the new image and a font is specified. The code then loops through a range of **roi_array** and checks if the speed limit at that index is greater than 0. This is required due to the fact that for "hard-negative mining" image vectors in the sample array supplied, the corresponding value in the array **category** was -1. If it is greater than 0, the code increments the **sign_counter** variable and draws a white rectangle on the new image at the specified coordinates. The speed limit is then added as text to the new image using the **draw.text** method, and the value of the speed limit is printed to the console.

```
# Create new ImageDraw object
draw = ImageDraw.Draw(img_with_text)
font = ImageFont.truetype("arial.ttf", 50)

# Counter for no. of actual speed signs present,
# as some ROIs are determined to not be a speed sign
sign_counter = 0

for i in list(range(len(roi_array))):
    # Image vectors used for "hard-negative mining" return a speed limit of -1
    # The following if-loop prevents them from producing a text label
    if speed_limits_arr[i] > 0:
        sign_counter+=1
        # Co-ordinates of corners of labels used to draw white rectangle
        draw.rectangle([box_corner1[i],box_corner2[i] ] , fill="white")
        # Text denoting speed limit added to image
        draw.text(text_loc[i], f"{int(speed_limits_arr[i])} km/h", font=font, fill=(255, 165, 0))
        print(f"Speed sign {sign_counter}: {int(speed_limits_arr[i])} km/h")
```

Figure 31 - Generation of text-labelled image

No sign present

The case of no legitimate speed sign being detected is dealt with using the following if-statement:

```
if sign_counter == 0:
    print("No signs detected!")
```

Figure 32 - No Sign Present Code

Image Display

Finally, the newly labelled image is displayed using the following Matplotlib commands:

```
# Show final image
plt.imshow(img_with_text)
plt.title(img_name)
plt.axis('off')
plt.show()
```

Figure 33 - Image display

Conclusions

Our project was quite successful in detecting speed signs and determining their speed limit values.

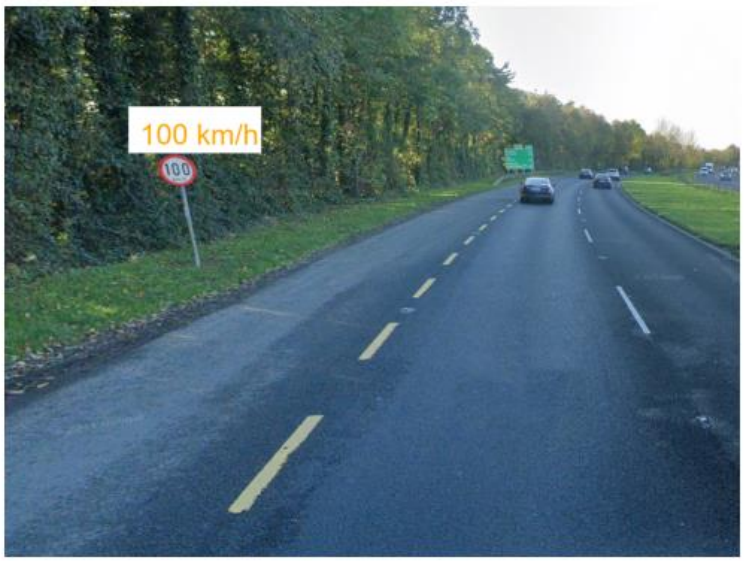
Out of the 32 speed signs present, 31 were correctly detected and classified, 1 was detected but incorrectly classified (80-004x2), and in 1 image (40-0002x1.png) a region that was not a speed sign was detected as an ROI and classified as a speed sign. All images with text labels and terminal responses are included in the Appendix.

This was a challenging but rewarding project. Challenges encountered included difficulties in finding the correct values for the HSV mask, generating our own k-NN classifier and trying to correctly overlay sign value labels onto the original images. All of these challenges were overcame through persistence and reviewing image processing notes and techniques which we came across in our lectures and various web resources.

Overall, this project greatly improved our understanding of image processing and the practical image processing methods implemented using Python programming.

Appendix

100-0001x1.png




```
Finding regions of interest in image 100-0001x1.png...
Bounding box of region 3: (267, 257, 336, 318)
ROI search complete...
Classifier complete...
Speed sign 1: 100 km/h
```

Figure 34 - Output for Image 100-0001x1.png

100-0002x1.png


```
Finding regions of interest in image 100-0002x1.png...
Bounding box of region 10: (206, 692, 366, 841)
ROI search complete...
Classifier complete...
Speed sign 1: 100 km/h
```

Figure 35 - Output for Image 100-0002x1.png

100-0003x1.png


```
Finding regions of interest in image 100-0003x1.png...
Bounding box of region 5: (1082, 603, 1168, 682)
ROI search complete...
Classifier complete...
Speed sign 1: 100 km/h
```

Figure 36 - Output for Image 100-0003x1.png

100-0004x1.png


```
Finding regions of interest in image 100-0004x1.png...
Bounding box of region 7: (945, 415, 1052, 534)
Bounding box of region 12: (945, 538, 1052, 657)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 100 km/h
```

Figure 37 - Output for Image 100-0004x1.png


```
Finding regions of interest in image 100-0005x1.png...
Bounding box of region 1: (203, 599, 267, 660)
ROI search complete...
Classifier complete...
Speed sign 1: 100 km/h
```

Figure 38 - Output for Image 100-0005x1.png


```
Finding regions of interest in image 120-0001x1.png...
Bounding box of region 1: (1090, 490, 1236, 645)
ROI search complete...
Classifier complete...
Speed sign 1: 120 km/h
```

Figure 39 - Output for Image 120-0001x1.png

120-0004x1.png


```
Finding regions of interest in image 120-0004x1.png...
Bounding box of region 3: (90, 325, 268, 517)
ROI search complete...
Classifier complete...
Speed sign 1: 120 km/h
```

Figure 40 - Output for Image 120-0004x1.png

120-0005x1.png


```
Finding regions of interest in image 120-0005x1.png...
Bounding box of region 1: (889, 194, 1053, 368)
ROI search complete...
Classifier complete...
Speed sign 1: 120 km/h
```

Figure 41 - Output for Image 120-0005x1.png

120-0002x2.png


```
Finding regions of interest in image 120-0002x2.png...
Bounding box of region 1: (1070, 441, 1142, 508)
Bounding box of region 2: (132, 469, 186, 528)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 120 km/h
Speed sign 2: 120 km/h
```

Figure 42 - Output for Image 120-0002x2.png

120-0003x1.png


```
Finding regions of interest in image 120-0003x1.png...
Bounding box of region 1: (987, 290, 1078, 384)
ROI search complete...
Classifier complete...
Speed sign 1: 120 km/h
```

Figure 43 - Output for Image 120-0003x1.png

40-0001x1.png


```
Finding regions of interest in image 40-0001x1.png...
Bounding box of region 17: (260, 576, 333, 632)
ROI search complete...
Classifier complete...
Speed sign 1: 40 km/h
```

Figure 44 - Output for Image 40-0001x1.png

40-0002x1.png


```
Finding regions of interest in image 40-0002x1.png...
Bounding box of region 1: (201, 1, 380, 157)
Bounding box of region 6: (126, 341, 274, 466)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 100 km/h
Speed sign 2: 40 km/h
```

Figure 45 - Output for Image 40-0002x1.png

NOTE: IN 40-0002x1.png THE WELCOME SIGN WAS PICKED UP AS AN ROI AND CATEGORISED AS 100 KM/H, BUT THE LABEL IS OFF SCREEN

50-0001x1.png


```
Finding regions of interest in image 50-0001x1.png...
Bounding box of region 16: (375, 307, 444, 379)
ROI search complete...
Classifier complete...
Speed sign 1: 50 km/h
```

Figure 46 - Output for Image 50-0001x1.png

50-0002x1.png


```
Finding regions of interest in image 50-0002x1.png...
Bounding box of region 1: (781, 493, 826, 545)
ROI search complete...
Classifier complete...
Speed sign 1: 50 km/h
```

Figure 47 - Output for Image 50-0002x1.png

50-0003x1.png


```
Finding regions of interest in image 50-0003x1.png...
Bounding box of region 40: (237, 368, 345, 472)
ROI search complete...
Classifier complete...
Speed sign 1: 50 km/h
```

Figure 48 - Output for Image 50-0003x1.png

50-0004x2.png


```
Finding regions of interest in image 50-0004x2.png...
Bounding box of region 12: (305, 694, 353, 749)
Bounding box of region 13: (1126, 700, 1164, 753)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 50 km/h
Speed sign 2: 50 km/h
```

Figure 49 - Output for Image 50-0004x2.png

50-0005x1.png


```
Finding regions of interest in image 50-0005x1.png...
Bounding box of region 2: (772, 632, 814, 685)
ROI search complete...
Classifier complete...
Speed sign 1: 50 km/h
```

Figure 50 - Output for Image 50-0005x1.png

60-0001x1.png


```
Finding regions of interest in image 60-0001x1.png...
Bounding box of region 10: (864, 386, 938, 455)
ROI search complete...
Classifier complete...
Speed sign 1: 60 km/h
```

Figure 51 - Output for Image 60-0001x1.png

60-0002x1.png


```
Finding regions of interest in image 60-0002x1.png...
Bounding box of region 9: (123, 248, 201, 316)
ROI search complete...
Classifier complete...
Speed sign 1: 60 km/h
```

Figure 52 - Output for Image 60-0002x1.png

60-0003x1.png


```
Finding regions of interest in image 60-0003x1.png...
Bounding box of region 4: (1037, 645, 1110, 712)
ROI search complete...
Classifier complete...
Speed sign 1: 60 km/h
```

Figure 53 - Output for Image 60-0003x1.png

60-0004x2.png


```
Finding regions of interest in image 60-0004x2.png...
Bounding box of region 1: (1059, 610, 1105, 660)
Bounding box of region 2: (143, 620, 207, 678)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 60 km/h
Speed sign 2: 60 km/h
```

Figure 54 - Output for Image 60-0004x2.png

60-0005x1.png


```
Finding regions of interest in image 60-0005x1.png...
Bounding box of region 5: (1052, 636, 1156, 711)
ROI search complete...
Classifier complete...
Speed sign 1: 60 km/h
```

Figure 55 - Output for Image 60-0005x1.png

80-0001x1.png


```
Finding regions of interest in image 80-0001x1.png...
Bounding box of region 11: (181, 243, 316, 376)
ROI search complete...
Classifier complete...
Speed sign 1: 80 km/h
```

Figure 56 - Output for Image 80-0001x1.png

80-0002x2.png


```
Finding regions of interest in image 80-0002x2.png...
Bounding box of region 2: (173, 407, 254, 472)
Bounding box of region 3: (1051, 428, 1102, 481)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 80 km/h
Speed sign 2: 80 km/h
```

Figure 57 - Output for Image 80-0002x2.png

80-0003x2.png


```
Finding regions of interest in image 80-0003x2.png...
Bounding box of region 6: (1205, 631, 1256, 695)
Bounding box of region 13: (46, 681, 145, 775)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 80 km/h
Speed sign 2: 80 km/h
```

Figure 58 - Output for Image 80-0003x2.png

80-0004x2.png


```
Finding regions of interest in image 80-0004x2.png...
Bounding box of region 10: (1096, 585, 1173, 668)
Bounding box of region 11: (219, 588, 294, 666)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 80 km/h
Speed sign 2: 60 km/h
```

Figure 59 - Output for Image 80-0004x2.png

80-0005x1.png


```
Finding regions of interest in image 80-0005x1.png...
Bounding box of region 1: (354, 385, 429, 452)
Bounding box of region 4: (350, 459, 435, 536)
ROI search complete...
Classifier complete...
Classifier complete...
Speed sign 1: 80 km/h
```

Figure 60 - Output for Image 80-0005x1.png

keep-left-0001x1.png

```
Finding regions of interest in image keep-left-0001x1.png...
ROI search complete...
No signs detected!
```

Figure 61 - Output for Image keep-left-0001x1.png