Contents

[Introduction 1](#_Toc21594818)

[Methodology 1](#_Toc21594819)

[Digit Classifier 1](#_Toc21594820)

[Images 1](#_Toc21594821)

[HOG Descriptors 2](#_Toc21594822)

[K-nearest-neighbour Model 2](#_Toc21594823)

[Saving and Loading 3](#_Toc21594824)

[Task 1 3](#_Toc21594825)

[Task Description 3](#_Toc21594826)

[Pipeline Overview 3](#_Toc21594827)

[Thresholding 3](#_Toc21594828)

[Noise Reduction 4](#_Toc21594829)

[Contouring 5](#_Toc21594830)

[Classification 5](#_Toc21594831)

[Spatial Grouping 5](#_Toc21594832)

[Validation Performance 5](#_Toc21594833)

[Task 2 5](#_Toc21594834)

[Pipeline Overview 5](#_Toc21594835)

[Feature Examples 5](#_Toc21594836)

[Validation Performance 5](#_Toc21594837)

[Conclusion 5](#_Toc21594838)

# Introduction

The task at hand is to extract a feature area from an image containing a building number or a set of building numbers and their directions as given by a sign post.

There are two tasks within this assignment, the first deals with a single building number and no directional sign, while the second requires the extraction of a list of building numbers and their directions according to the signage. Each task will require a pipeline of pre-processing, feature detection, feature extraction, feature classification and finally an output in human readable format.

# Methodology

For both tasks, a generally similar approach has been taken due to key similarities in the tasks at hand. Both tasks have sets of three digits that are aligned on the horizontal plane and consist of white digits on a black background. The sample signage and validation sets only include white digits on black background signs, so little effort has been made to ensure the following pipelines can handle signage outside this scope.

A considerable effort has been made to get both tasks to classify all of the signage examples with 100% accuracy and this was achieved, however small changes do lead to a decrease in classification accuracy meaning that the methods used are likely to be somewhat overfit to the data provided.

Number and directional arrow classification for this assignment required both classification and an accuracy measure which was well suited to an implementation of a K-nearest-neighbour classifier that could provide a pseudo accuracy measure by Euclidean distance between then digit to be classified and the K other samples it matched with. All digit classification for both tasks in this assignment are handled by the python class “classifier.py” that contains an implementation of KNN classifier with a K value of five and that was trained on 450 samples of each digit and arrow augmentation as were provided with the assignment. The details of the classification method are discussed further in the following section.

For both tasks, a series of consecutively smaller regions of interest within an image have been established. First, the image is loaded in where the general area of the black background plate is acquired. Once this is found, the image is binarized with a threshold and then OpenCV’s contouring algorithm is applied to isolate blobs that are potential digits or arrows. Each of these potential blobs are first filtered based on width to height ratios and area constraints before being run through the classifier where the best matches are selected to be used in the next stage of the classification pipeline. In the case of Task 1, the classified digits are assessed for their special proximity while in Task 2, positive arrow matches are used to further refine the signpost location. The specifics of each tasks’ mythology and performance is discussed in their respective sections.

# Digit Classifier

The digit classifier is a K-nearest-neighbour model classifier trained on the HOG descriptors of the digits 0-9 and the left and right arrows (🡨 and 🡪). ADD MORE

## Images

The images in this classifier must be the same size for both training and classification. The sample set of digits provided by the assignment contains five of each digit 0-9 and the two arrows that are not mutated and a mutated set with 500 samples of each digit and arrow. These images are all 28 pixels wide by 40 pixels tall except for one of the arrows which is 29 pixels wide and is truncated to conform to the 28x40 standard set. The images must all be the same size, so they have the same number of HOG descriptors.

As HOG is not scale invariant, each of the images to be classified must be scaled down to 28x40 before it can be classified. At first, images were scaled to this size directly, but due to the exacting nature of the feature extraction step prior, the digits were often misclassified as the digits to be classified had foreground pixels touching all borders. This caused misclassification as the training set of images mainly consisted of digits that were not touching any edges as shown in Figure 1. To combat this, the images to be classified were scaled to the size 24x36 and a black border of two pixels wide was added. This drastically improved classification accuracy.



Figure 1 - A mutated training set eight

The method of image resizing also had a large impact on classification accuracy, with the default linear interpolation method having a worse classification rate than a selected one. If the image was larger than the desired 24x36 downscale size, the classifier used the Area Interpolation method and if the image was smaller than the desired size, Bicubic Interpolation is used. These provided better results than the default method.

## HOG Descriptors

The Histogram of Oriented Gradients (HOG) descriptors are formed by sliding a HOG cell block of size 14x20 across the image in half steps such that there is a total of nine overlapping locations on the digit where HOG is performed. This sliding action of the HOG block over the full image and division into nine angular bins results in a HOG description vector of size 1x324. Each of the training digits has a HOG vector computed for it and saved with a corresponding label.

## K-nearest-neighbour Model

The K-nearest-neighbour model uses a majority classification based on the most prevalent classification of the K nearest neighbours in the N-dimensional space. In this case, the model is a 324-dimensional space, with each dimension for one of the HOG descriptor vector values. A K value of five was chosen primarily through experimentation, however a K value of seven worked similarly well.

The Knn method in OpenCV returns the classification ID and an array of distances to each of the K nearest elements from the one to be classified. The sum of these distances was used as a pseudo classification accuracy measure. An image that is not a digit will still return from the Knn with a digit ID, but the sum of distances will be very large as it is not close to a region where real digits reside in the 324-diemsnional space.

For this reason, lower values of K were not as stable, as a larger K value is what gave the classification confidence of what was being classified, and the sum of distances ruled out noise and false positives through thresholding. A threshold for a true positive digit match was found to be anything with a sum of distances less than 3.0 x K. This value and method will be brought up again in the sections for Task 1 and Task 2, as both tasks make use of this classification accuracy.

## Saving and Loading

When the initialisation function in the classifier class is called, it checks the relative directory “/model/” for four saved descriptor and label files. These files are “traning.npy”, “training\_labels.npy”, “test.npy” and “test\_labels.npy”. These files contain the precomputed HOG descriptors for sets of the training digits and their corresponding labels in Numpy array binary files. If these files exist, they are read, and the training dataset and training labels are loaded into the Knn model. If they do not exist however, the training set of 450 samples and test set of 50 samples is generated, saved and then loaded into the Knn model.

This was a design choice to save computation time on start-up and to reduce the size of required files that are included with the program.

Figure 2 - An example input image for Task 1



# Task 1

## Task Description

In Task 1, a set of images in a folder is to be loaded in and for each, the three-digit building number extracted and saved in both text and image format. The program must return exactly one region per input image, containing only the building number. There are no directional arrows in this task, however the classifier used still has these arrows trained in it. The numbers are white, large font numbers on a rectangular black plaque with an example being shown in Figure 2.

## Pipeline Overview

For this task, the image is passed into the task 1 method as a colour image as loaded by OpenCV where it is then converted to greyscale. The greyscale image is passed through an adaptive thresholding method as provided by OpenCV with a binary type threshold and then a median filter is applied to denoise the edges. Contouring is applied to the image after thresholding to outline enclosed boundaries. Each of these closed contours has the rectangular bounding box for it returned and the area within this bounding box is sent to the classifier class if it satisfies area and aspect ratio constraints. The classifier returns the classified digit along with sum of distances for that digit’s classification and then smallest sum of distance digits are shortlisted for spatial classification.

The spatial classifier orders the ten best digits by area and then by Y coordinate, looping over all iterations of combinations of three digits. This triplet grouping is only valid if the Y coordinate, area and digit height are within a certain threshold of each other. All valid groups found in the given set of ten digits is saved, along with the sum of sum of distances for the three digits that make it up. The best signage match is the valid triplet with the smallest sum of sum of distances. The digits from the best triplet are ordered by X coordinate and then the sign’s text and region of interest are saved to their appropriate files.

## Thresholding

There were three possibilities considered for the thresholding in Task 1, including binary thresholding, Otsu’s thresholding and OpenCV’s adaptive thresholding. The primary objective for thresholding the input image was to isolate the digits as foreground against the black plaque as background. From testing, Otsu’s threshold on an entire input image would often return a threshold result that was unusable for digit separation, especially when the black background plate of the sign was not close to a true black colour. An example of each of the attempted thresholding types is shown below.

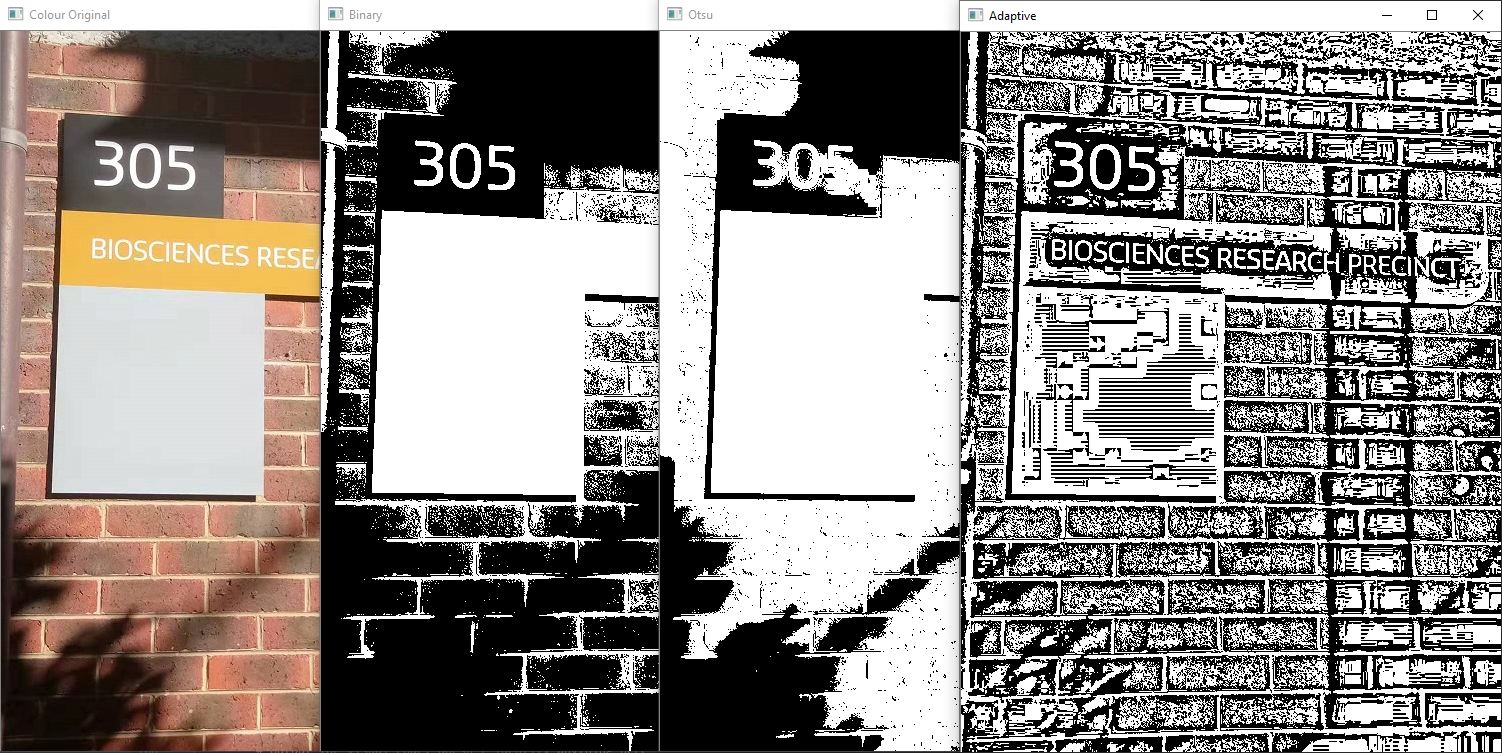


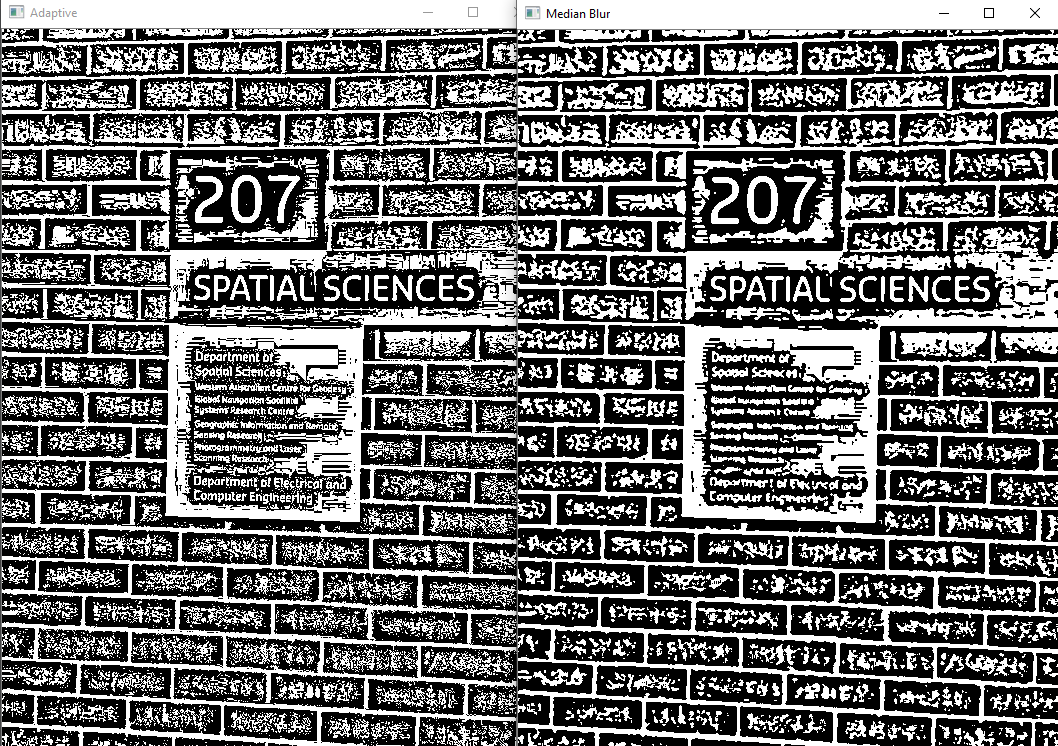
Figure 3 - Original Task 1 Image; Binary Threshold; Otsu Threshold; Adaptive Threshold

Binary thresholding on the entire image was more consistent than Otsu’s method but often suffered in accuracy as the threshold given was fixed between input images and failed to get the desired separation on digits that were in shadow. This fixed thresholding value was varied between 100 and 160 but nothing worked across the entire set of test and validation input images.

The adaptive thresholding method provided by OpenCV, that applies a dynamic threshold on very small sections of the image, provided the desired result of each digit in all input images. Regardless of the digit’s contrast to the local background, adaptive thresholding isolated them with an unbroken border of background pixels that was essential to the contouring algorithm. An adaptive thresholding local area of size 21 was used and its efficacy can also be seen on the building’s text in Figure 3.

## Noise Reduction

Figure 4 - Before and after Median Filtering on the adaptive threshold result

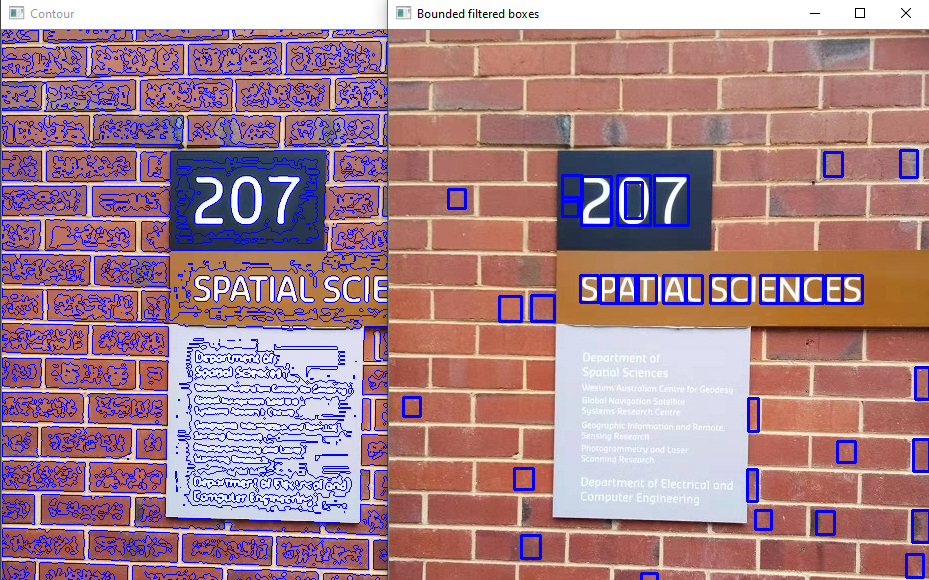


The median filter noise reduction step may seem pointless, but its effect was to increase clustering so that the contouring step would not take as long to compute without sacrificing sharp edges. A median filter width of three pixels approximately halved the number of contours found and saved a significant amount of computational time that had flow on effects for the rest of the pipeline. Figure 4 gives an example of how effectively the ‘static’ in the left-hand image is merged.

## Contouring

Contouring and digit classification make up much of the classification pipeline for Task 1. OpenCV’s findContours() method in CHAIN\_APPROX\_SIMPLE mode has been used to isolate blobs and mark potential digits. This method finds a simple approximation of the outline of features on the binarized image and for each of these features the bounding box is calculated.

Figure 5 - Raw Contours (Left). Feature filtered bounding boxes for contours (right)



Attempting to classify every single bounding box generated would be very inefficient so, instead, only bounding boxes that met predetermined characterises are attempted. To be classified, a feature must have a bounding box of more than 300 pixels in area and have a height to width ratio between 1.1 and 4.0. For the bounding boxes that satisfy the precondition, the corresponding region of interest from the original greyscale image is passed to the classifier.

As an example shown in Figure 5, each of the bounding boxes on the right-hand image will be sent to the classifier to get be classified with a digit and a confidence value, being the sum of distances. Everything within the bounds of the bounding box in the corresponding greyscale image is sent to the classifier and this inclusion of contextual information helped reduce the false positive rate.

## 



Figure 6 – Digits classified and ranked by confidence. Green is a stronger match

## Classification

The classifier methodology user for each of the bounding boxes used in this task has been described fully in the Digit Classifier section. The list of all classified bounding boxes returned from the classifier is ranked by confidence related to the sum of distances given by the digit classifier method. An example scale of confidence is shown in Figure 6 ranging from high confidence in green to low confidence in red. The colour of the bounding box has been setup for display purpose only. Of these newly classified digits, the top ten highest confidence value boxes will be used in spatial classification. If there are less than ten total boxes, all of them are grouped spatially.

## Spatial Grouping

‘Spatial Grouping’ here will refer to a collection of tests and thresholds that work primarily on the physical proximity and shape of the bounding boxes so that they best approximate what would be found on a correctly classified sign. Traits such as digit height, area and Y coordinate are all similar for each of the three digits on the building number signs.

The spatial grouping method tries all possible combinations of three boxes to find valid sets of three digits that will be referred to as triplets. A valid triplet is one where all digits have Y coordinates within 20 pixels, an area that is no more than 50% different and a height that is no more than 40% different. To do this, each of the possible pairs within a triplet are checked such that the difference in a trait is no more than the difference threshold multiplied by the digit with the larger value for that trait. It has not been overlooked that this is a rather slow and inefficient operation that can easily approach a time complexity of O(N2) but it has been deemed a non-issue as the number of digits, N, is ten.

## Output Generation

The output for Task 1 is required to be one image file of the region of interest and one text file containing the building number as text. For the text output, the three digits within the triplet are ordered by ascending X coordinate and read of in left to right format to give the three-digit building number. For the region of interest image, the three digits are again sorted by ascending X coordinate and a bounding region between the left digit’s top-left corner and the last digit’s bottom-right corner is copied from the original colour input image and saved with OpenCV’s imwrite().

## Validation Performance

Table - Task 1 Validation image results

|  |  |  |
| --- | --- | --- |
| File | Classified Number | Region of Interest |
| Test01.jpg | “Building 202” |  |
| Test02.jpg | “Building 314” |  |
| Test03.jpg | “Building 301” |  |
| Test04.jpg | “Building 109” |  |
| Test05.jpg | “Building 206” |  |
| Test06.jpg | “Building 312” |  |

This implementation of Task 1 classifies all test images, validation images and additional input images with 100% accuracy. From the supplied validation files, the regions and text extraction for each of the six test images, Test01.jpg to Test06.jpg, are shown in Table 1.

# Task 2



Figure 7 - An example input image for Task 2

## Task Description

In Task 2, the input image with contain a directional signpost that has multiple sets of three digits, each with a directional arrow either pointing left or right. To complete this task for a given input image, the program must isolate the region of interest of all the digits on the signpost and output a text file with each of the three-digit building number and their direction labelled either “left” or “right”. It has been assumed that there will only be one signpost, and it will consist of white digits on a black background. An example input image provided for Task 2 is shown here.

## Pipeline Overview

The classification pipeline for this task is mode involved than the one descripted in Task 1 and happens in four major steps. The first objective is to approximately locate and isolate the signpost along the horizontal direction. This step is a rough approximation of the location and uses Sobel gradients to locate the ‘flat’ texture of the sign. The second step is to use contours and classification to find the arrows on the sign, and from that, realign the sign using a perspective transform.

Step three takes the cropped sign region and finds the arrows using contouring on it for a second time, and from this, extrapolates where the three numbers related to that arrow are. These sets of digits and arrows will be called digit regions. The fourth step is to take each of the digit regions and classify the digits out in order.

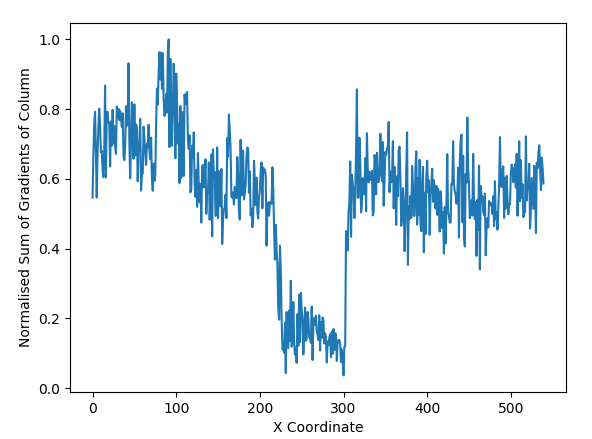


Figure 8 - Normalised sum of gradients in image columns

## Horizontal Position Approximation

The horizontal position of the sign can be approximated by summing the squares of the Sobel gradient by columns in the image and finding a cluster of columns with the minimum gradient sum. The logic behind this is that the signposts are almost always positioned against a natural background which has a lot of variation and noise, while the sign itself has large areas of relatively constant intensity.

Figure 9 - Example of a sign in an approximated crop

Figure 8 shows a plot of normalised sum of gradients in each column, with the signpost being located between 220 and 300 along the X axis. As a human this is simple to make a best guess as to where the sign might be, but the data is very noisy and variant so a single threshold will not work. Instead, a sweeping threshold algorithm has been designed to gradually increase the threshold from 0.0 until more than a specified percentage of the X range is covered between the maximum and minimum X coordinates included by the threshold. The maximum and minimum X from one step before this range threshold was breached are used to crop the image and approximate the location of the sign as demonstrated in Figure 9.

## Locating the Sign



Figure 10 - Region outputs to locate the sign

After the sign’s position has been approximated, the region containing the numbers must be found. To do this, an adaptive threshold is applied over the image with a blocksize of 25 and then this image is contoured to find each of the regions. The method here is the same as used in Task 1 and both binary and Otsu’s thresholding methods do not separate the digits from the background.

Next, all regions that have a height to width ratio between 0.6 and 1.4, and an area greater than 40 pixels are sent to be classified by the digit classifier. All positive matches for arrows with a sum of errors less than 15 are added to the list.

Figure 10 shows a good example of this process working. The yellow bounded regions are arrows that met the height to width ratio, the area and the sum of errors thresholds. The blue bounding boxes also met the initial criterial but were either not classified as an arrow or exceeded the sum of errors (confidence) value. The green bounding boxes are generated for each of the arrows and are 4.9 times the width of their arrow and 2.2 times the height of their arrow, with alignment as shown. These relationships were found through trial and error and showed to be consistent across the testing set of input images.

## Cropping the Sign

A four-point perspective transform is primarily used to straighten up and crop the sign after the arrows on the sign have been located. The initial set of points to be used with OpenCV’s getPerspectiveTransform() method are the top two corners of the top green bounding box and the bottom two corners of the bottom green bounding box. These are the numbers 201 and 208 on the sign in Figure 10.

The final points for the perspective transform are made up of the maximum width and height of the input region and the interpolation method chosen was the bicubic method to get the best quality transform from OpenCV’s warpPerspective() method.

## Grouping the Digits



Figure - Digit groupings outlined in green

The steps in the Locating the Sign section are repeated here on the fully cropped sign with tighter tolerances on the values. The cropped signage area is doubled in size which lead to an increase in accuracy for the classification of arrows. For a feature to be classified, it must have a height to width ratio between 0.80 and 1.20 as well as a minimum area that is proportional to the width of the sign.

Figure 11 shows the progression of this step and all the identified arrows are now much more uniform in size and shape. Initially, the finding arrows and extracting the digits was done once but that method did not work as well on smaller signs or signs with a larger viewing angle.

The digit regions generated from the arrows here are the full width of the cropped sign and 2.0 times the height of the arrow. There is still noise on the left hand side of many of these digit regions but this will be removed in the same step that all the digits for each digit region are classified.

## Classification of Digits

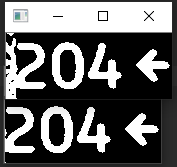


Figure - Digit region trimming

Each digit region should now contain three digits and an arrow, with a high likelihood of noise on the left-hand end of the region. This noise is removed by classifying a single digit’s width of the image alone and selecting the region that has the best confidence or lowest sum of errors. This is then used to crop the noise off the left-hand side of the region. A before and after example is shown in Figure 12.

Next, the image is split between the digits, using the sum of intensities in a column to determine where a digit starts and stops. Digits are separated by scanning left to right. Each separate digit is classified within its digit region and appended together to form the building number and direction, e.g. “204R”. This code will be appended to a list and returned to the controlling code that handles file writing.

## Validation Performance

# Conclusion

# Appendix 1 – Task1\_final.py