

Machine Perception Assignment



11 OCTOBER 2019

ADRIAN SHEDLEY, 19142946

Contents

Introduction	1
Methodology.....	1
Digit Classifier.....	1
Images	1
HOG Descriptors	2
K-nearest-neighbour Model.....	2
Saving and Loading	2
Task 1	3
Task Description.....	3
Pipeline Overview	3
Thresholding	3
Noise Reduction	4
Contouring	5
Classification	5
Spatial Grouping.....	6
Output Generation.....	6
Validation Performance	6
Task 2	7
Task Description.....	7
Pipeline Overview	7
Horizontal Position Approximation.....	7
Locating the Sign	8
Cropping the Sign.....	8
Grouping the Digits	8
Classification of Digits	9
Validation Performance	9
Conclusion.....	10
Appendix 1 – classifier.py	11
Appendix 2 – task1.py.....	14
Appendix 3 – run_task1.py	17
Appendix 4 – task2.py.....	18
Appendix 5 – run_task2.py	23

Introduction

There are two tasks within this assignment, the first deals with a single building number and no directional arrow, while the second requires the extraction of a list of building numbers and their directions. Each task requires a pipeline of preprocessing, feature detection, feature extraction, feature classification and finally output in a 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 axis and consist of white digits on a black background. The sample signage and validation sets only include digits of this type, so little effort has been made to ensure the following pipelines can classify signage outside this scope.

A considerable effort has been made to get both tasks to classify all input signage examples with 100% accuracy and this was achieved. Additional input images were used and achieved 100% classification accuracy, however, it is still possible that the pipelines were overfitted.

Number and directional arrow classification for this assignment required both classification and an accuracy measure. This was well suited to an implementation of a K-nearest-neighbour classifier that could provide a pseudo accuracy measure by Euclidean distance between the digit to be classified and the K other samples it was matched with. All digit classification for both tasks in this assignment is handled by the python class “classifier.py” that contains an implementation of a Knn classifier. 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 has been established, contoured, classified and verified. While similar, Task 1 cannot be reused within TASK 2 due to slight differences in the requirements of the task. The specifics of each tasks’ methodology and performance is discussed in depth in their respective sections.

Digit Classifier

The digit classifier is a K-nearest-neighbour classifier trained on the HOG descriptors of the digits 0-9 and the left and right arrows (← and →). The class “classifier.py” must be initialised once before use on program start-up. For simplicity, the classifier returns an ID for each classification such that digits 0-9 correspond to 0-9, a left arrow is 10 and a right arrow is 11.

Images

The images in this classifier needed to be the same size for both training and classification. The sample set of digits provided by the assignment contained five of each digit 0-9 and the two arrows that were not mutated as well as a mutated set with 500 samples of each digit and arrow. These images were 28 pixels wide by 40 pixels tall except for one of the arrows which was 29 pixels wide and was truncated to conform to the 28x40 standard set. The images needed to be the same size so they would have the same number of HOG descriptors. The classifier was trained on a final set of 450 of the provided mutated images for the digits.

As HOG is not scale-invariant, each of the images to be classified needed to be scaled down to 28x40 before it could 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 they had foreground pixels touching all borders. This misclassification was caused by a nuance in the training set of images that 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 case-specific one. If the image was larger than the desired 24x36 size, the classifier used the Area Interpolation method and if the image was smaller than the desired size, Bicubic Interpolation was used. These provided better results than the default method.

HOG Descriptors

The Histogram of Oriented Gradients (HOG) descriptors were formed by sliding a HOG cell block of size 14x20 across the image in half steps so that there were a total of nine overlapping locations on the digit. This sliding action of the HOG block over the full image and division into nine angular bins resulted in a HOG description vector of size 1x324. Each of the 450 training digits had a HOG vector computed for it and it was saved along with a corresponding label to file.

K-nearest-neighbour Model

The K-nearest-neighbour (Knn) model uses a majority classification based on the K nearest neighbours in an N-dimensional space. In this case, the model used 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 returned 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 confidence measure, with a lesser error being a higher confidence. An image that was not a digit would still return from the classifier with a digit ID, but the sum of distances would be large as it was not close to a region where the training digits resided in the 324-dimensional space.

Lower values of K were not as stable when classifying noisy data but 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 $3.0 \times K$. 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 to pick true positives.

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 `"training.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 a 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 the required files to be included with the program.

Task 1

Task Description

In Task 1, a set of images in a folder was to be loaded 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 were no directional arrows in this task, however, the classifier used still has these arrows trained in it. The numbers were white, large font numbers on a rectangular black plaque with an example shown in Figure 2.

Pipeline Overview

For this task, the image was 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 was applied to denoise the edges. Contouring was applied to the image after thresholding to outline enclosed boundaries. Each of these closed contours had the rectangular bounding box for it generated and the area within this bounding box was sent to the classifier class if it satisfied area and aspect ratio constraints. The classifier returned the classified digit along with a sum of distances for that digit's classification.

The spatial classifier ordered the 10 best digits by area and then by Y coordinate, looping over all iterations of combinations of the three digits, called a triplet. This triplet grouping was only valid if the Y coordinate, area and digit height were within a certain threshold of each other. All valid groups found in the given set of 10 digits was saved, along with the sum of sum of distances for the three digits that made it up. The best signage match was the valid triplet with the smallest sum of sum of distances. Finally, 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 pixels against the black plaque as background pixels. 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 true black colour. An example of each of the attempted thresholding types is shown below.



Figure 2 - An example input image for Task 1



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 applied 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 for isolation can also be seen on the building's text in Figure 3.

Noise Reduction

The median filter's 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 into clusters.

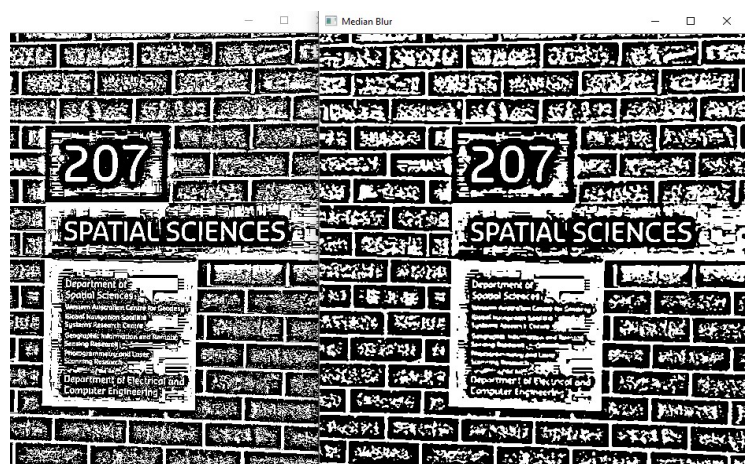
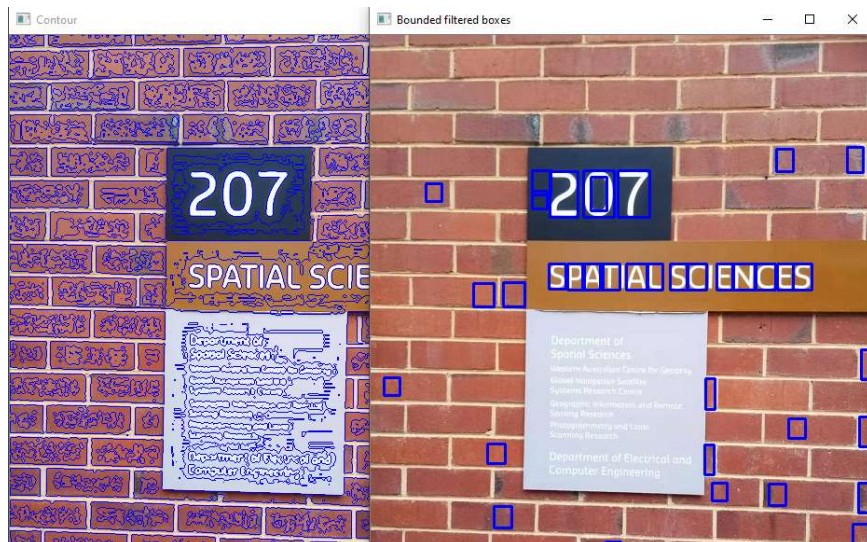


Figure 4 - Before and after Median Filtering on the adaptive threshold result

Contouring

Contouring and digit classification made up much of the classification pipeline for Task 1. OpenCV's `findContours()` method in `CHAIN_APPROX_SIMPLE` mode was used to isolate blobs and mark potential digits. This method found a simple approximation of the outline of features on the binarized image and for each of these features from which the bounding box was calculated.



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

As shown in Figure 5, each of the

bounding boxes on the right-hand image met this precondition and would be sent to the classifier to be classified with a digit and a confidence value (sum of distances). Everything within the bounds of the bounding box in the corresponding greyscale image was sent to the classifier and this inclusion of contextual information helped reduce the false positive rate.

Classification

The classifier methodology used for each of the bounding boxes in this task has been described fully in the Digit Classifier section. The list of all classified bounding boxes returned from the classifier was ranked by confidence as 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 set up for display purpose only. Of these newly classified digits, the top 10 highest confidence value boxes would be used in the spatial grouping. If there were less than 10 total boxes, all of them were grouped spatially.



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

Spatial Grouping

'Spatial Grouping' here refers to a collection of tests and thresholds that worked primarily on the physical proximity and shape of the bounding boxes that best approximated what would be found on a correctly classified sign. Traits such as digit height, area and Y coordinate were all similar for each of the three digits on the building number signs.

The spatial grouping method attempted all possible combinations of three boxes to find valid sets of three digits that are referred to as triplets. A valid triplet was one where all digits have Y coordinates within 20 pixels, an area that was no more than 50% different and a height that was no more than 40% different. To do this, each of the possible pairs within a triplet was checked such that the difference in a trait was no more than the difference threshold multiplied by the digit with the larger value for that trait. It was not overlooked that this was a slow and inefficient operation that could easily approach a time complexity of $O(N^2)$ but it was deemed a non-issue as the number of digits, N, was 10.







Output Generation

The output for Task 1 was 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 were ordered by ascending X coordinate and read in left to right format to give the three-digit building number. For the region of interest image, the three digits were 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 was copied from the original colour input image and saved with OpenCV's `imwrite()`.

Validation Performance

This implementation of Task 1 classified 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.

Table 1 - 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"	

Task 2

Task Description

In Task 2, the input image contained a directional signpost that had 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 numbers 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.



Figure 7 - An example input image for Task 2

Pipeline Overview

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

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

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 were almost always positioned against a

natural background which had a lot of variation and noise, while the sign itself had large areas of relatively constant intensity.

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

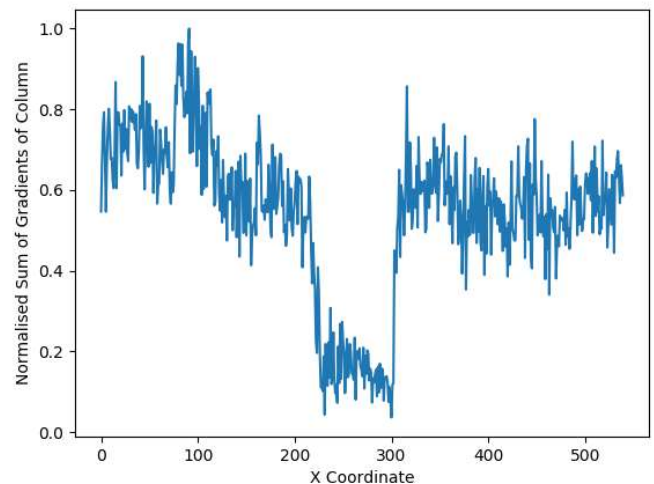


Figure 8 - Normalised sum of gradients in image columns



Figure 9 - Example of a sign in an approximated crop

Locating the Sign

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

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

Figure 10 shows a good example of this process's output. 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 criteria but were not classified as an arrow or exceeded the sum of errors (confidence) value. The green bounding boxes were generated for each of the arrows and were 4.9 times the width of their arrow and 2.2 times the height of the 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.

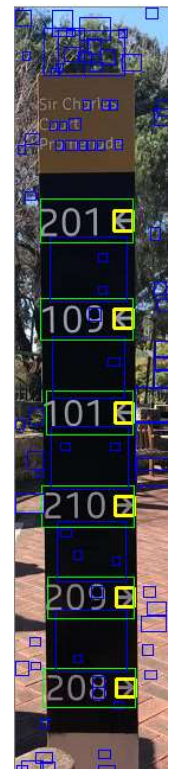


Figure 10 - Region outputs to locate the sign

Cropping the Sign

A four-point perspective transform was primarily used to straighten up and crop the sign after the arrows on the sign had been located. The initial set of points to be used with OpenCV's `getPerspectiveTransform()` method were the top two corners of the top green bounding box and the bottom two corners of the bottom green bounding box. The use of this method was inspired by an example from pyimgsearch.com (Rosebrock 2014).

The final points for the perspective transform were derived from 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. Other methods significantly blurred the cropped and transformed sign which made further classification difficult.

Grouping the Digits

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 was doubled in size which led 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 was proportional to the width of the sign.

Figure 11 shows the progression of this step and all the identified arrows were much more uniform in size and shape. Initially, the finding arrows and extracting the digits was done once only, 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 were the full width of the cropped sign and 2.0 times the height of the arrow. There was still noise on the left-hand side of many of these digit regions but this was to be removed in the same step that all the digits for each digit region are classified.



Figure 11 - Digit groupings outlined in green

Classification of Digits

Each digit region now contained three digits and an arrow, with a high likelihood of noise on the left-hand end of the region. This noise was removed by classifying a single digit's width of the image from the left-hand side, sliding across one pixel and repeating. From these iterations, the region that had the best confidence or lowest sum of errors was the true start of the image. This was then used to crop the noise off the left-hand side of the region. A before and after example is shown in Figure 12.



Figure 12 - Digit region trimming

Next, the image was split between the digits, using the sum of intensities in a column to determine where a digit started and stopped. Digits were separated by scanning left to right and grouping columns where the sum of intensities was greater than zero. The digits were cropped once more to remove blank space above and below the digit to better fit the input of the classifier. Figure 13 shows the result for a set of individual cropped digits before they were sent to the classifier. Again, the classifier used was the one described in the section Digit Classifier.





Figure 13 - Isolated digits from a digit region

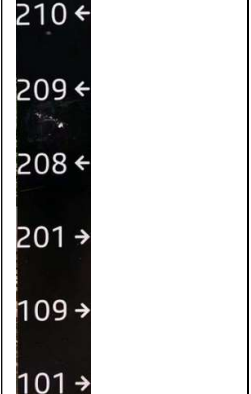
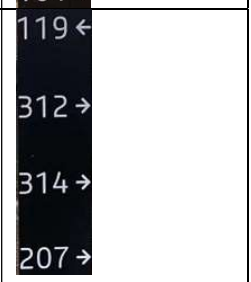
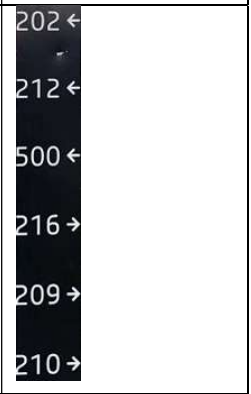
Each separate digit was classified within its digit region and appended together to form the building number and direction, e.g. "204R". This code was appended to a list and returned to the controlling code that handles file writing.

Validation Performance

The results from the validation set of images for Part 2 yielded a 100% success rate in all digits and directions on all signs. Each of the extracted regions and their text outputs are tabulated below.

Table 2 - Task 2 Validation Files and Results

Task 2 File	Classified Numbers and Direction	Region of Interest
	"Building 208 to the left Building 501 to the left Building 205 to the right Building 202 to the right Building 206 to the right"	208 ← 501 ← 205 → 202 → 206 →
	"Building 303 to the left Building 305 to the left Building 300 to the left Building 301 to the right Building 312 to the right Building 309 to the right"	303 ← 305 ← 300 ← 301 → 312 → 309 →

		<p>“Building 210 to the left Building 209 to the left Building 208 to the left Building 201 to the right Building 109 to the right Building 101 to the right”</p>	
		<p>“Building 119 to the left Building 312 to the right Building 314 to the right Building 207 to the right”</p>	
		<p>“Building 202 to the left Building 212 to the left Building 500 to the left Building 216 to the right Building 209 to the right Building 210 to the right”</p>	
		<p>“Building 302 to the left Building 311 to the left Building 204 to the right Building 599 to the right Building 201 to the right”</p>	

Conclusion

The classification pipelines for both tasks perform as well as they can on the sets of input data provided. The methods used are laid out in a logical and sequential order, however, multiple thresholds and feature parameters embedded within each task have been tweaked to match this dataset alone and are thus overfit.

Appendix 1 – classifier.py

```
# Author Adrian Shedley, 19142946
# Date: 4 oct 2019
# This is the image classifier that contains a kNN and a HOG descriptor that will handle most of the
heavy lifting in
# terms of classifying possible regions of interest from the feature extractor.
```

```
import os
import os.path
import cv2
import numpy as np
from os import listdir
from os import path as pth

# Running environment variables
IMG_PATH = '/res/'
TRAIN_VAR_PATH = '/res/training/augmented'

TRAIN_SET = 'model/training.npy'
TRAIN_LABELS = 'model/training_labels.npy'
TEST_SET = 'model/test.npy'
TEST_LABELS = 'model/test_labels.npy'

# HOG descriptor parameters
FEAT_WIDTH = 28
FEAT_HEIGHT = 40
WIN_SIZE = (FEAT_WIDTH, FEAT_HEIGHT) # (28, 40)
BLOCK_SIZE = (FEAT_WIDTH // 2, FEAT_HEIGHT // 2) # (14, 20)
BLOCK_STRIDE = (FEAT_WIDTH // 4, FEAT_HEIGHT // 4) # (7, 10)
CELL_SIZE = (FEAT_WIDTH // 4, FEAT_HEIGHT // 4) # (7, 10)
N_BINS = 9
HOG_FEATURES = 324

# kNN Model parameters
NUMBERS = ['Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine',
'LeftArrow', 'RightArrow']
FEATURES = 12
TRAIN_FEATURES = 490
TEST_FEATURES = 5
K = 5

# HOG descriptor and kNN initial setup
hog_des = cv2.HOGDescriptor(WIN_SIZE, BLOCK_SIZE, BLOCK_STRIDE, CELL_SIZE, N_BINS)
knn = cv2.ml.KNearest_create()

def init(show_test=False):
    """Initialise, load training files"""
    if not (pth.exists(TRAIN_SET) and pth.exists(TRAIN_LABELS) and pth.exists(TEST_SET) and
pth.exists(TEST_LABELS)):
        # one or more of the needed files for the kNN model are missing, recreate and save them
        train()

    # Reload the data from file (cleanest way even if we just made it)
    training = np.load(TRAIN_SET)
    training_labels = np.load(TRAIN_LABELS)

    # Train the kNN with the loaded data
    knn.train(training, cv2.ml.ROW_SAMPLE, training_labels)

    if show_test:
        test_set = np.load(TEST_SET)
        test_labels = np.load(TEST_LABELS)
        _, result, _, _ = knn.findNearest(test_set, k=K)
        result = result.reshape(FEATURES * TEST_FEATURES)
        print('kNN classifier for augmented numbers with HOG: ', accuracy(test_labels, result))

def train():
    """ Perform Knn digit set building and calculation of HOG features"""
    training_aug = list()
    # Initiate the array to have a List willing to accept each of the Augmented images as pixels
    for i in range(len(NUMBERS) + 1):
```

```

        training_aug.append(list())

# Load the augmented images in greyscale
path = os.getcwd() + TRAIN_VAR_PATH
all_folders = listdir(path)
for folder in all_folders:
    # Go inside each of the translation folders
    for augm in listdir(path + '/' + folder):
        # If the file is a jpg, attempt to load it
        if augm.endswith(".jpg"):
            img = cv2.imread(path + '/' + folder + '/' + augm, 0)
            if img is not None:
                # Match the image title to an internal index for that number using the dictionary
                for i, nn in enumerate(NUMBERS):
                    if augm.startswith(nn):
                        # Once we have found what this digit was, add it to the list and move on
                        training_aug[i].append(img[:FEAT_HEIGHT, :FEAT_WIDTH])
                        break

# Create the datasets (HOG Descriptors) and labels for training and testing
train_set, train_label = training_hog(training_aug)
test_set, test_label = test_hog(training_aug)

#numpy save it to file for use with different runs
np.save(TRAIN_SET, train_set)
np.save(TRAIN_LABELS, train_label)
np.save(TEST_SET, test_set)
np.save(TEST_LABELS, test_label)

def classify(img):
    """ Given a greyscale image img, return a classification ID and sum of errors (confidence)"""

    if img.shape[0] > 0 and img.shape[1] > 0:
        border = 2
        method = cv2.INTER_AREA

        # Decide if the image is upscaled or downscaled, and then select an interpolation method
        if img.shape[0] < 40 or img.shape[1] < 28: # Upscale
            method = cv2.INTER_CUBIC

        # Resize the image to (28, 40) with the content scaled to (22, 34) and a black border of 2
        img_resize = cv2.resize(img, (FEAT_WIDTH - border * 2, FEAT_HEIGHT - border * 2),
            interpolation=method)
        # Add a border width 2 of black
        img_resize = cv2.copyMakeBorder(img_resize, border, border, border, border,
            cv2.BORDER_CONSTANT)
        vect = hog(img_resize).reshape(1, HOG_FEATURES)
        _, result, _, dist = kNN.findNearest(vect, k=K)
        return int(result[0][0]), dist
    else:
        # Size 0 image, will crash, return false classify code ie 13
        return FEATURES + 1, 100

def hog(img):
    return hog_des.compute(img)

def training_hog(master):
    """Build a set of training features and labels"""
    return subset_hog(0, FEATURES, TRAIN_FEATURES, master)

def test_hog(master):
    """Build a set of test features and labels"""
    return subset_hog(TRAIN_FEATURES, FEATURES, TEST_FEATURES, master)

def subset_hog(low, digits, samples=100, master_set=None):
    """Subset the augmented training digits"""
    set = np.zeros((samples * digits, HOG_FEATURES), dtype=np.float32)
    set_labels = np.zeros((samples * digits), dtype=int)
    matrixAug = np.array(master_set)

```

```

# Loop over each type of digit, and get enough samples from each
for i in range(digits):
    for j in range(samples):
        # Copy the greyscale
        im = matrixAug[i][low + j]
        # Perform HOG
        hh = hog(im)
        # reshape to 1x324
        set[i * samples + j] = hh.reshape(HOG_FEATURES)
        # Save the HOG
        set_labels[i * samples + j] = int(i)

return set, set_labels

def accuracy(labels, result):
    # Compare all of the provided labels with the classified labels to get a training percentage
    matches = result == labels
    correct = np.count_nonzero(matches)
    accuracy = correct * 100.0 / result.size
    return accuracy

```

Appendix 2 – task1.py

```
# Author Adrian Shedley
# date: 4 Oct 2019
# Machine Perception task 1 assignment, code has been commented

import numpy as np
import cv2
import classifier as cl

# Initialise the Classifier before anything else
cl.init()

def spatially_similar(roi1, roi2):
    """Returns true when two features are not the same feature, have similar Y coordinates, similar
    area and a similar
    height value. These are set byt the absolute Y threshold in pixels, an area scale factor and a
    height scale factor.
    The default values for these have been set by experimentation"""
    Y_THRESH = 20
    AREA_THRESH = 0.5
    HEIGHT_THRESH = 0.4
    return (abs(roi1.y - roi2.y) <= Y_THRESH) and (abs(roi1.area - roi2.area) <= max(roi1.area,
    roi2.area) *
    AREA_THRESH) and (abs(roi1.h - roi2.h) <= max(roi1.h, roi2.h) * HEIGHT_THRESH) and (roi1
    != roi2)

def order_features(feats):
    """Returns the 3 numbers in a given sign in left to right format according to their X
    coordinate"""
    nums = list()
    for i in range(3):
        nums.append(feats[i])

    # Sort by smallest X coordinate
    nums.sort(key=lambda dig: dig.x)
    return nums

def numbers(feats):
    """Returns the building number as a 3 digit number from left to right"""
    nums = order_features(feats)
    return str(nums[0].classify) + str(nums[1].classify) + str(nums[2].classify)

def extracted_area(img, feat):
    """Returns an area from the Top-Left corner of the left digit to the bottom-right corner of the
    right digit"""
    nums = order_features(feats)
    # nums[0] is left digit, nums[2] is right digit
    img_cropped = img[nums[0].y: nums[2].y + nums[2].h, nums[0].x: nums[2].x + nums[2].w]
    return img_cropped

def spatial_grouping(rois):
    # Sort by smallest area then by smallest X value
    rois.sort(key=lambda dig: dig.area)
    rois.sort(key=lambda dig: dig.x)

    temp = list()

    for i, el in enumerate(rois):
        for j, el2 in enumerate(rois[i:]):
            for k, el3 in enumerate(rois[j:]):
                if spatially_similar(el, el2) and spatially_similar(el, el3) and
                spatially_similar(el2, el3):
                    temp.append([el, el2, el3, el.error + el2.error + el3.error])

    # Sort by smallest error sum of all 3 features ie index [3]
    temp.sort(key=lambda x: x[3])
    return temp[0]
```



```

# Take an image of a building sign and get the numbers
def task1(img, name=None):
    if img is not None:
        # Convert to greyscale
        grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Thresholding with the adaptive thresholding method
        thresh = cv2.adaptiveThreshold(grey, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY,
        blockSize=21, C=1)
        # Median blur to clump groups together
        thresh = cv2.medianBlur(thresh, 3)
        # Invert so that the numbers are foreground and can be contoured
        invert = np.uint8(thresh * -1 + 255)

        # Compute all the contours on the inverted thresholded image
        _, contours, heir = cv2.findContours(invert, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        #im4 = img.copy()
        #cv2.drawContours(im4, contours, -1, (255, 0, 0), 1)
        #cv2.imshow('Contour', im4)
        im3 = img.copy()

        # Allocate a list to store all of the potential digits bounding boxes and classifications in
        digit_list = list()

        # Loop over all the contours, and for those that meet the Area and Aspect Ratio requirements,
        classify them
        for i in contours:
            x, y, w, h = cv2.boundingRect(i) # Get the (x,y) and width and height of a bounding box
            for this contour
                if 300 < w*h < 10000:
                    if 1.1*w < h < 4*w:
                        # Save a region of interest from the greyscale image within the bounding box
                        roi = grey[y : y + h, x : x + w]
                        # Get the classification and confidence (errors sum) for this ROI
                        classify, errors = cl.classify(roi)
                        # Create a new Digit class that carries a neat representation of the data
                        digit = Digit(classify, np.sum(errors), x, y, w, h, w*h)
                        # Add the digit to the digit list that will be sorted and spatially grouped
                        digit_list.append(digit)

                        #cv2.putText(im3, str(classify), (x, y - 5), cv2.FONT_HERSHEY_SIMPLEX, 1, (0,255-
10*int(np.sum(errors)), 10*int(np.sum(errors))), 2)
                        #cv2.rectangle(im3, (x, y), (x + w, y + h), (0,255-10*int(np.sum(errors)),
10*int(np.sum(errors))), 2)

                        #cv2.imshow('Ranked Classifications', im3)
                        #cv2.waitKey(0)

                        # Sort by the best confidence first (Smallest Error)
                        digit_list.sort(key=lambda x: x.error)
                        # After sorting, take the top 10 best digits by confidence and return the best triple
                        features = spatial_grouping(digit_list[:min(10, len(digit_list))])

                        # Output for visual code, doesn't identify anything here. Take the 3 digits from the triplet
                        and draw them
                        for el in features[:3]:
                            cv2.rectangle(im3, (el.x, el.y), (el.x + el.w, el.y + el.h), (0, 255, 0), 2)
                            cv2.putText(im3, str(el.classify), (el.x, el.y - 5), cv2.FONT_HERSHEY_SIMPLEX, 1, (120,
255, 0), 2)

                        # Return the numbers as a string, and the Region Of Interest they were found in
                        return numbers(features), extracted_area(img, features)
                    else:
                        # Image is none. Return an error
                        return "???", np.zeros((1,1))

class Digit:
    def __init__(self, classify, error, x, y, w, h, area):
        """This is a data storage class for a classified digit.
        The Digit class contains a digits classification, confidence (error), x pos, y pos
        width, height and precomputed area. """
        self.classify = classify

```

```
self.error = error # Also the confidence measure
self.x = x
self.y = y
self.w = w
self.h = h
self.area = area
```

Appendix 3 – run_task1.py

```
# Author Adrian Shedley, Date 6 Oct 2019
# Purpose of this file is to complete the task 1 as per the assignmnet spec sheet
# It also assumes that the directories exist

# READ ALL FILES FROM /home/student/test/task1/
# FILE NAMES ARE: testX.jpg
# REQUIRES OUTPUT: DetectedAreaX.jpg and BuildingX.txt

import cv2
import task1 as task1
import os
from os import listdir

TASK1_FILES = "/home/student/test/task1/"
TASK1_OUTPUT = os.getcwd() + "/output/task1/"

def main():
    all_files = listdir(TASK1_FILES)
    for file in all_files:
        if file.startswith("test") and file.endswith(".jpg"):
            number = file[4:-4]
            img = cv2.imread(TASK1_FILES + file)
            if img is not None:
                num, roi = task1.task1(img)

                # Output region
                cv2.imwrite(TASK1_OUTPUT + "DetectedArea" + number + ".jpg", roi)

                #Output text file
                f = open(TASK1_OUTPUT + "Building" + number + ".txt", "w+")
                f.write("Building " + num + "\n")
                f.close()

    print("Complete")

if __name__ == "__main__":
    main()
```

Appendix 4 – task2.py

Author Adrian Shedley
date 5 oct 2019

```
import numpy as np
import cv2
import classifier as cl
import matplotlib.pyplot as plt

cl.init()

def normalise(A):
    """Normalise a matrix"""
    return (A - A.min()) / (A.max() - A.min())

def thresh_sweep(arr, step, inclusion):
    """Sweep a threshold up from 0,0 by step until inclusion percentage of the image is included"""
    # The range of X values before thresholding stops
    range = int(arr.shape[0] * inclusion)
    arr2 = arr.copy()

    # Sweep the threshold
    for thresh in np.arange(0.0, 1.0, step):
        arr2[arr2 < thresh] = 0.0
        # find the min and max included X values
        where = np.argwhere(arr2 == 0)
        # Test that the range is not exceeded
        if where.shape[0] != 0 and (where.max() - where.min()) > range:
            arr2 = arr.copy()
            # Go one step back, and reset the threshold
            arr2[arr2 < thresh - step] = 0.0
            arr2[arr2 >= thresh - step] = 1.0
            where = np.argwhere(arr2 == 0)
            arr2[where.min():where.max()] = 0.0
            break

    # return the lower and upper X values
    return where.min(), where.max()

def crop_height(img):
    """Take a greyscale image and remove the top and bottom black bars"""
    mini = 0
    maxi = img.shape[0]

    # Loop over the rows from top down, until a pixel of non zero is found
    for i, row in enumerate(img):
        if np.sum(row) > 0:
            mini = max(0, i - 1)
            break

    # Loop from bottom up until a row of non-zero is found
    for i, row in enumerate(np.flip(img, axis=0)):
        if np.sum(row) > 0:
            maxi = img.shape[0] - (i - 1)
            break

    # return the cropped image
    return img[max(0, mini-1):min(img.shape[0], maxi+1), :]

def number_block(col_sum, start_point=0, reverse=False, mini=3):
    """Segment the next block of non-zero columns from an image. Default from the left to right
    operation
    block must consist of at least mini columns and starts from start_point"""
    start = 0
    stop = 0
    starting = start_point
    ending = col_sum.shape[0]
    step = 1
    # If we are moving from the right hand side, reverse is true
    if reverse:
        starting = col_sum.shape[0]-1
```



```

        ending = start_point+1
        step = -1
    # find the start of the block
    for i in range(starting, ending, step):
        if col_sum[i] > 0:
            start = i
            break
    # Find the end of the block
    if reverse:
        starting = col_sum.shape[0]-1
        ending = start+1
    else:
        starting = start
        ending = col_sum.shape[0]
    # If we have not yet exceeded the minimum number of columns included
    for i in range(starting, ending, step):
        if col_sum[i] == 0 and i - start > mini:
            stop = i
            break
        else:
            stop = i
    # returning the start and stop X values depending on the operating mode
    if reverse:
        return stop, start
    else:
        return start, stop

def classify_digits(region, digits=None, reverse=False):
    """Classifies all 3 digits and an arrow in a digit region"""
    region = crop_height(region) # Shrink the region
    # Sum everything in columns
    col_sum = np.sum(region, axis=0)

    if digits is None:
        digits = region.copy()

    # Section each digit based on the black columns, each start and stop is a digit.
    start1, stop1 = number_block(col_sum, 0, reverse)
    start2, stop2 = number_block(col_sum, stop1 + 1, reverse)
    start3, stop3 = number_block(col_sum, stop2 + 1, reverse)
    start4, stop4 = number_block(col_sum, stop3 + 1, reverse)

    #Extract the digits from the image. Num4 is the arrow
    num1 = digits[:, start1:stop1]
    num2 = digits[:, start2:stop2]
    num3 = digits[:, start3:stop3]
    arrow = digits[:, start4:stop4]

    # Crop the digits down vertically
    num1 = crop_height(num1)
    num2 = crop_height(num2)
    num3 = crop_height(num3)
    arrow = crop_height(arrow)
    # Return the classification ID's for each of the regions.
    return cl.classify(num1)[0], cl.classify(num2)[0], cl.classify(num3)[0], cl.classify(arrow)[0]

def crop_sign(img_rgb, img_grey, digit_regions):
    """Combined function that takes an ordered list of digit regions and crops and perspective
    transforms the
    image so that the top two corners of the first region and the bottom two corners of the last
    region are the
    extents of a newly created rectangular image."""
    # Methodology inspired by 4 Point OpenCV getPerspective Transform Example:
    # REF: https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/

    # Make region of interest to save (straight)
    treg = digit_regions[0] # Top of sign Region
    breg = digit_regions[-1] # Bottom of sign region (the last element in the list)

    # Starting at the top left corner of the sign, going clockwise
    pt1 = [treg.rx, treg.ry]
    pt2 = [treg.rx + treg.rw, treg.ry]

```

```

pt3 = [breg.rx + breg.rw, breg.ry + breg.rh]
pt4 = [breg.rx, breg.ry + breg.rh]
initial_pts = np.float32([pt1, pt2, pt3, pt4])

# Calculate the size that the output image will become after perspective warp
new_height = abs(max(pt1[1] - pt4[1], pt2[1] - pt3[1])) # Difference in left side height and
right side height
new_width = abs(max(treg.rw, breg.rw)) # Largest of top width and bottom width
final_pts = np.float32([[0, 0], [new_width - 1, 0], [new_width - 1, new_height - 1], [0,
new_height - 1]])

# Generate a perspective transform matrix that maps initial_pts to final_pts
M = cv2.getPerspectiveTransform(initial_pts, final_pts)
warped_grey = cv2.warpPerspective(img_grey, M, (new_width, new_height), flags=cv2.INTER_CUBIC)
warped_rgb = cv2.warpPerspective(img_rgb, M, (new_width, new_height), flags=cv2.INTER_CUBIC)

return warped_rgb, warped_grey

def validate_classify(num1, num2, num3, num4):
    """Validate whether or not the 4 numbers given are [DIGIT DIGIT DIGIT ARROW] format. Returns false
    if not"""
    valid = True
    # If the first three nums are between 0-9
    if num1 > 9 or num2 > 9 or num3 > 9:
        valid = False

    # if num4 (the arrow) is left arrow (10) or right arrow (11)
    if num4 < 10 or num4 > 11:
        valid = False

    return valid

# Take an image of a directional sign and get the numbers
def task2(img, name=None):
    if img is not None:
        # Convert the image to greyscale
        grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Get the sobel gradient on both the X and Y directions, and sum them together
        sobX, sobY = cv2.spatialGradient(grey, None, None, 3)
        sob = np.square(sobX) + np.square(sobY)
        # Sum down each of the columns (axis=0)
        col_sum = np.sum(sob, axis=0)
        col_sum = col_sum / col_sum.max() # Normalise between 0 and 1.0

        # Get the Lower and Upper X coordinates where an approximation of the sign lies
        minX, maxX = thresh_sweep(col_sum, 0.001, 0.22)

        # Extract a working area that is a little wider than the sign's best match area, ensuring
        valid bounds
        roi = grey[:, max(0, int(minX*0.9)):min(grey.shape[1], int(maxX*1.1))]
        roi_rgb = img[:, max(0, int(minX*0.9)):min(grey.shape[1], int(maxX*1.1))]

        # Adaptive threshold the macro ROI and then apply contouring
        roi_thresh = cv2.adaptiveThreshold(roi, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY, 25, 1)
        _, contours, heir = cv2.findContours(roi_thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

        # First pass classification to find the ARROWS on the sign only. This step is to align the
        sign
        digit_regions = list()
        for cont in contours:
            # Return the (x,y) and width and height of a bounding rectangle for this contour
            x, y, w, h = cv2.boundingRect(cont)
            # Only consider regions that are the correct shape and aspect initially.
            if 40 < w * h < 5000 and 0.6 * w < h < 1.4 * w:
                # Get the underlying greyscale region for this bounding box for classification
                possible_digit = roi[y:y + h, x:x + w]
                classify, errors = cl.classify(possible_digit)
                # If the confidence of this digit is below the threshold and it is a Left arrow (10)
                or right arrow (11)
                if np.sum(errors) < 3.0 * cl.FEATURES and (int(classify) == 10 or int(classify) ==

```

```

11):
    #cv2.rectangle(roi_rgb, (x, y), (x+w, y+h), (0, 255, 255), thickness=2)
    # Calculate the full region with digits and arrow from the size of the arrow
    rx = max(0, x - int(3.9 * w))
    ry = max(0, y - int(0.6 * h))
    rw = int(4.9 * w)
    rh = int(2.2 * h)
    # Create a Digit Region and populate the data
    dr = DigitRegion(rx, ry, rw, rh, x, y, w, h, int(classify))
    digit_regions.append(dr)
    #cv2.rectangle(roi_rgb, (rx, ry), (rx + rw, ry + rh), (0, 255, 0), thickness=1)
#else:
    #cv2.rectangle(roi_rgb, (x, y), (x+w, y+h), (255, 0, 0), thickness=1)

# Sort the list by the Y Coordinate
digit_regions.sort(key=lambda dr: dr.ry)

# Crop the sign down to only include the arrows and numbers
region_output, cropped = crop_sign(roi_rgb, roi, digit_regions)

# Upscale the image by 2 times and adaptive threshold it.
cropped = cv2.resize(cropped, (cropped.shape[1] * 2, cropped.shape[0] * 2), cv2.INTER_LINEAR)
up_thresh = cv2.adaptiveThreshold(cropped, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY, 21, 1)
# Contour the upscaled threshold image
_, contours, heir = cv2.findContours(up_thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

# 3 channel greyscale for drawing coloured rectangles on
warped_rgb = cv2.cvtColor(cropped, cv2.COLOR_GRAY2BGR)

# Make a new list to store the new warped regions
digit_regions = list()
# Find the digit regions based on the location and size of the arrows
for cont in contours:
    x, y, w, h = cv2.boundingRect(cont)
    # If the region area is within the threshold and that the width height ratio is
acceptable.
    if (np.power(cropped.shape[1], 2))//40 < w * h < 5000 and 0.8 * w < h < 1.2 * w:
        possible_digit = cropped[y: y + h, x: x + w]
        classify, errors = cl.classify(possible_digit)
        # See if the digit is classified as an arrow and is sufficiently low in error. 10 =
left, 11 = right
        if np.sum(errors) < 3.0 * cl.FEATURES and (int(classify) == 10 or int(classify) ==
11):
            cv2.rectangle(warped_rgb, (x, y), (x+w, y+h), (0, 255, 255), thickness=2)
            rx = 0 # Use the full sign width
            ry = max(0, y - int(.5 * h))
            rw = cropped.shape[1] # Using the full sign width
            rh = int(2.0 * h)
            # Make it a DigitRegion and add it to the list
            dr = DigitRegion(rx, ry, rw, rh, x, y, w, h, classify)
            digit_regions.append(dr)
            cv2.rectangle(warped_rgb, (rx, ry), (rx + rw, ry + rh), (0, 255, 0), thickness=1)
        else:
            cv2.rectangle(warped_rgb, (x, y), (x+w, y+h), (255, 0, 0), thickness=1)

# Order the List by ascending region Y coordinate
digit_regions.sort(key=lambda dr: dr.ry)

# Allocate a list to store the text on the sign, eg "113R"
numbers_on_sign = list()

# For each of the regions, sweep in from the left hand side, checking which gives the best
classification,
# which indicates the true edge of the sign has been found.
for dr in digit_regions:
    digits = cropped[dr.ry: dr.ry + dr.rh, dr.rx: rx + dr.rw]
    digits = cv2.threshold(digits, 120, 255, cv2.THRESH_OTSU)[1]

    width = int(dr.w * 1.2)
    # Noise reduction matches. Find the best image classification from left hand side inward
    bests = list()
    for sweep in range(0, width//2):
        part = digits[:, sweep:width]

```

```

        part = crop_height(part)
        #cv2.imshow('part', part)
        classify, errors = cl.classify(part)
        bests.append([classify, np.sum(errors*errors), sweep])

    # sort the best matches by their lowest distance classification
    bests.sort(key=lambda x: x[1])

    # get the coordinate of the best digit, and crop the digit region
    digits = digits[:, int(bests[0][2]):]

    # first try left to right full height classification
    num1, num2, num3, num4 = classify_digits(digits)

    # Multiple attempts if for some reason the numebtrs returned were illogical. E.g. an arrow
    where a digit is
    if not validate_classify(num1, num2, num3, num4):
        # Try only the top 70% of the digits image, all digits still form blocks here
        num1, num2, num3, num4 = classify_digits(digits[:int(digits.shape[0]*0.70), :],
digits)

    #if not validate_classify(num1, num2, num3, num4):
        # Try the full image in back to front order
        num1, num2, num3, num4 = classify_digits(digits, reverse=True)

    #if not validate_classify(num1, num2, num3, num4):
        #num1, num2, num3, num4 = classify_digits(digits[:int(digits.shape[0] * 0.70), :],
digits, reverse=True)

    # Save the output string of numbers
    output = str(num1) + str(num2) + str(num3)

    dir = num4
    # If the direction is Left
    if dir == 10:
        output += "L"
    elif dir == 11: # if it is right
        output += "R"

    # Add the numbers to the list of directions on this sign
    numbers_on_sign.append(output)

    # return the numbers on the sign as a list and as a region
    return np.array(numbers_on_sign), region_output
else:
    # Not a valid image, return error
    return "????", np.zeros((1, 1))

class DigitRegion:
    """A data storage class for the region containing 3 digits and an arrow"""
    def __init__(self, rx, ry, rw, rh, x, y, w, h, classify):
        self.rx = rx
        self.ry = ry
        self.rw = rw
        self.rh = rh
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.classify = classify

```


Appendix 5 – run_task2.py

```
# Author Adrian Shedley, Date 6 Oct 2019
# Purpose of this file is to complete the task 1 as per the assignment spec sheet
# It also assumes that the directories exist

# READ ALL FILES FROM /home/student/test/task2/
# FILE NAMES ARE: testX.jpg
# REQUIRES OUTPUT: DetectedAreaX.jpg and BuildingX.txt

import cv2
import task2 as task2
import os
from os import listdir

TASK2_FILES = "/home/student/test/task2/"
TASK2_OUTPUT = os.getcwd() + "/output/task2/"

def main():
    all_files = listdir(TASK2_FILES)
    for file in all_files:
        if file.startswith("test") and file.endswith(".jpg"):
            number = file[4:-4]
            img = cv2.imread(TASK2_FILES + file)
            if img is not None:
                nums, roi = task2.task2(img)

                # Output region
                cv2.imwrite(TASK2_OUTPUT + "DetectedArea" + number + ".jpg", roi)

                #Output text file
                f = open(TASK2_OUTPUT + "Building" + number + ".txt", "w+")
                for num in nums:
                    actual_num = num[:3]
                    direct = num[-1:]
                    directional = "left"
                    if direct == 'R':
                        directional = "right"
                    f.write("Building " + actual_num + " to the " + directional + "\n")
                f.close()

    print("Complete")

if __name__ == "__main__":
    main()
```