Image Classification and Number Recognition

Lachlan Ross 18836523

Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Ross	Student ID:	18836523
Other name(s):	Lachlan		
Unit name:	Machine Perception	Unit ID:	COMP3007
Lecturer / unit coordinator:	Senjian An	Tutor:	Sam Bray
Date of submission:	12/10/19	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Lachlan Ross Date of signature: 12/10/19

(By submitting this form, you indicate that you agree with all the above text.)

Task 1 Code:

main.py :

This file is responsible for taking in a number of files, and running other modules to return the sign numbers and sign area, getting the predicted values of those numbers.

Then writing the sign area and predicted numbers to file

```
import glob
import cv2
from classification import runSetup
from classification import evaluateNumbers
from numberExtraction import runExtract

#grabs all jpg images in directory - directory path
files = glob.glob("home/student/test/task1/*.jpg")

#used when creating filenames to write to
count = 1

#trains svm algorithm on training set
model = runSetup()

#loops through each image to perform operations
for f in files:

    #extracts sign number images, and sign image
    retnum,sign,valid = runExtract(f)
    if(valid):

        #runs number images with classifier, which returns predicted
numbers
        ret = evaluateNumbers(model, retnum)

        #writes detected building numbers to file
        f = open("./output/Task1/Building"+str(count)+".txt", "w+")
        f.write("Building " + str(ret))

        #writes image sign detected to file
        cv2.imwrite('./output/Task1/BuildingSign'+str(count)+'.jpg',sign)

        count += 1
```

numberExtraction.py :

This file is responsible for taking in a coloured image, filtering and processing it to find the building sign, then extracting the building sign and each number inside of it

```
import cv2
import glob
from matplotlib import pyplot as plt
import numpy as np


#puts coordinate points for image rotation in a consistant order
def order_points(pts):
    #initialize ordered coordinates representing a rectangle, rectangle can
be angled
    #order, top left, top right, bottom right, bottom left,
    rect = np.zeros((4,2),dtype ="float32")


    #finds the minimum and maximum sum of coordinates as this represents
top left and bottom right
    minimum = pts[0]
    maximum = pts[0]
    for i in pts:
        if(i[0] + i[1] > maximum[0] + maximum[1]):
            maximum = i
        if(i[0] + i[1] < minimum[0] + minimum[1]):
            minimum = i

    rect[0] = minimum
    rect[2] = maximum

    #find the difference between coordinate values, as this represents top
right vs bottom left
    diff = np.diff(pts, axis = 1)
    rect[1] = pts[np.argmin(diff)]
    rect[3] = pts[np.argmax(diff)]
    return rect


#takes in a bounding box around the numbers on the building sign, and
rotates to remove tilt from numbers
def warpRotation(img, pts):
    #orders the coordinates consistantely
    rect = order_points(pts)
    (tl, tr, br, bl) = rect
```

```
    #calcs maximum possible width
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))

    #calcs max possible height
    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))

    dst = np.array([[0, 0],[maxWidth - 1, 0],[maxWidth - 1, maxHeight -
1],[0, maxHeight - 1]], dtype = "float32")

    #use getPerspective and warpPerspective to rotate rectangle straight
    M = cv2.getPerspectiveTransform(rect,dst)
    warped = cv2.warpPerspective(img, M, (maxWidth, maxHeight))
    return warped

#responsible for detecting the sign contour, post image processing
def find_squares(img):

    squares = []
    for gray in cv2.split(img):
        for thrs in range(0, 255, 26):
            if thrs == 0:
                bin = cv2.Canny(gray, 0, 50, apertureSize=5)
                bin = cv2.dilate(bin, None)
            else:
                retval, bin = cv2.threshold(gray, thrs, 255,
cv2.THRESH_BINARY)
            bin, contours, hierarchy = cv2.findContours(bin, cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)
            for cnt in contours:

                #implementation of douglas pecker algorithm, grabs a
polygon, but allows some leeway based on the arclength
                cnt_len = 0.08 * cv2.arcLength(cnt, True)
                cnt = cv2.approxPolyDP(cnt, cnt_len, True)
                #checks polygon has 4 sides, and is above and below size
thresholds
                if len(cnt) == 4 and cv2.contourArea(cnt) > 4000 and
cv2.contourArea(cnt)<30000 and cv2.isContourConvex(cnt):
                    squares.append(cnt)

    return squares


#processes and extracts numbers from sign
```

```
def process_sign(crop):

    badcontours = []

    height, width = crop.shape

    #grabs all contours using convex hull and puts minrect around them
    cropimg, contours, hierarchy = cv2.findContours(crop, cv2.RETR_LIST,
cv2.CHAIN_APPROX_SIMPLE)
    for cnt in contours:
        appended = False
        hull = cv2.convexHull(cnt)
        rect = cv2.minAreaRect(cnt)
        box = cv2.boxPoints(rect)
        box = np.int0(box)

        #if any rectangle section touches the edge, that contour is
considered bad
        if(box[0][0] < 1 or box[0][1] < 1 or box[1][0] < 1 or box[1][1] < 1
and box[2][0] < 1 or box[2][1] < 1 or box[3][0] < 1 or box[3][1] < 1):
            badcontours.append(cnt)
            appended = True

        if(box[0][0] >= width-1 or box[1][0] >= width-1 or box[2][0] >=
width-1 or box[3][0] >= width-5):
            if(not appended):
                badcontours.append(cnt)
                appended = True

        if(box[0][1] >= height-1 or box[1][1] >= height-1 or box[2][1] >=
height-1 or box[3][1] >= height-1):
            if(not appended):
                badcontours.append(cnt)
                appended = True

        #any contours under a certain size are considered bad
        if(cv2.contourArea(hull)<300 and not appended):
            badcontours.append(cnt)

    #bad contours are filled with black (0,0,0)
    cv2.fillPoly(cropimg, badcontours,(0,0,0))

    #cv2.imshow('aftercrop',cropimg)
    #cv2.waitKey(0)



    #grab remaining contours, should only be numbers now
    cropimg, contours, hierarchy = cv2.findContours(crop, cv2.RETR_LIST,
```

```
cv2.CHAIN_APPROX_SIMPLE)
    combine = []

    #takes the min area around combined contours, this gets angle of number
slant
    for cnt in contours:
        for i in cnt:
            combine.append(i)

    combine = np.asarray(combine)


    minrect = cv2.minAreaRect(combine)
    box = cv2.boxPoints(minrect)
    box = np.int0(box)

    #rotates slanted numbers to be straight
    cropimg = warpRotation(cropimg, box)

    #thresholds again after rotation reintroduces gray values
    _,cropimg = cv2.threshold(cropimg, 200, 255, cv2.THRESH_BINARY)

    #adds a thin border around number cluster
    cropimg =
cv2.copyMakeBorder(cropimg,10,10,10,10,cv2.BORDER_CONSTANT,(0,0,0))

    #grab individual external contours, each should be a number
    cropimg, contours, hierarchy = cv2.findContours(cropimg,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    numbers = []
    sort = []
    sortcont = []

    #place rectangle around each contour and grabs coordinate x value
representing top left coordinate
    for cnt in contours:
        hull = cv2.convexHull(cnt)
        x,y,w,h = cv2.boundingRect(cnt)
        sort.append(x)

    #implement bubble sort, based on x bounding box of each contour, to
sort left to right
    swapped = True
    while swapped:
        swapped = False
        for x in range(len(sort) -1):
            if sort[x] > sort[x+1]:
                sort[x], sort[x+1] = sort[x+1], sort[x]
```

```
                contours[x], contours[x+1] = contours[x+1], contours[x]
                swapped = True

    #place rectangle around each individual number, crop them out
seperately into a list
    for cnt in contours:
        hull = cv2.convexHull(cnt)
        x,y,w,h = cv2.boundingRect(cnt)

        crop = cropimg[y : y+h, x: x+w]

        crop = cv2.copyMakeBorder(crop,4,4,4,4,cv2.BORDER_CONSTANT,(0,0,0))

        numbers.append(crop)

    #return cropped numbers, and cropped signarea
    return numbers, cropimg


#main of this module, takes in 1 image to process at a time, locates sign
and extracts numbers
def runExtract(inFile):

#image preprocessing begins

#creating kernel used to sharpen images, and default values
    ddepth = -1
    sharpen = np.array((
          [0,-1,0],
          [-1,5,-1],
          [0,-1,0]), dtype ="int")

    #read in image
    img = cv2.imread(inFile)

    #blur, sharpen, blur
    img = cv2.GaussianBlur(img,(5,5),0)
    img = cv2.filter2D(img, ddepth, sharpen)
    img = cv2.GaussianBlur(img,(5,5),0)

    #convert to HSV
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    # define colour range, below 50 saturation defines white, gray and
black range.
    lower_val = np.array([0,0,0]) #lowest threshold is [0,0,0] black
    upper_val = np.array([179,50,255]) # all 179 hue values, all 255
values, 50/255 saturation
```

```
    #define black tinged with blue colour range
    lower_val2 = np.array([80,0,0])
    upper_val2 = np.array([150,255,100])


    # define red value range below 100 light level, 2 range sets are
required due to red being 0 on hsv gradient, up to 90 saturation
    lower_val3 = np.array([0,0,0])
    upper_val3 = np.array([15,90,100])
    lower_val4 = np.array([165,0,0])
    upper_val4 = np.array([179,90,100])


    #all light levels above 190 are set to white 255
    lower_val7 = np.array([0,0,190]) #lowest threshold is [0,0,0] black
    upper_val7 = np.array([179,50,255]) # all 179 hue values, all 255
values, 50/255 saturation


    #preprocess, by whitening and blackening some background sections
    mask = cv2.inRange(hsv, lower_val2, upper_val2)
    img[mask != 0] = [0,0,0]
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    mask = cv2.inRange(hsv, lower_val3, upper_val3)
    img[mask != 0] = [0,0,0]
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    mask = cv2.inRange(hsv, lower_val4, upper_val4)
    img[mask != 0] = [0,0,0]
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)


    #threshold using previously defined colour range
    #anything higher than the 50 saturation value will be converted to 115
gray, anything higher than 190 light level will be pure white
    mask = cv2.inRange(hsv, lower_val, upper_val)
    mask2 = cv2.inRange(hsv, lower_val7, upper_val7)
    mask = cv2.bitwise_not(mask)
    img[mask != 0] = [115,115,115]
    img[mask2 !=0] = [255,255,255]


    #make hsv processed image gray
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    #binarize on threshold 114
    _,thresh1 = cv2.threshold(gray, 114, 255, cv2.THRESH_BINARY)  # ensure
binary
```

```
    #create morphology kernel
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))

    #apply morphology kernel for the OPEN operation
    prebound = cv2.morphologyEx(thresh1, cv2.MORPH_OPEN, kernel)
    thresh1 = prebound

    #find contours of decent size and shape
    squares = find_squares(thresh1)
    if(len(squares) >= 1):
        rect = cv2.minAreaRect(squares[0])
        box = cv2.boxPoints(rect)
        box = np.int0(box)
        #cv2.drawContours(thresh1,[box],0,(100,100,100),2)
        x,y,w,h = squares[0]

        #the x,y,w,h values are different depending on the angle of square,
this checks to make sure the right values are used while cropping
        if(x[0][0] > w[0][0]):
            crop = thresh1[h[0][1]:y[0][1], h[0][0]:y[0][0]]
        else:
            crop = thresh1[x[0][1]:w[0][1], x[0][0]:w[0][0]]

        #if sign processing fails e.g.(sign detected, but numbers are bad),
catch expection here and return invalid
        try:
            x,y = process_sign(crop)
        except Exception:
            print("Sign Proccessing Failed")
            return False, False, False

        return x,y,True
    else:
        print("No Sign Detected")

    return False,False,False
```

classification.py :

```
import glob
import cv2
import sys
import numpy as np
import itertools as it

#grabs training set of images
files = glob.glob("./Digits/original/*.jpg")

# load training digits, label them appropriately
def load_digits(imglist):
    digits = []
    labels = []

    #label training set based on filename
    for img in files[0:]:
        digits_img = cv2.imread(img, 0)

        if('Zero' in img):
            digits.append(digits_img)
            labels.append(0)
        elif('One' in img):
            digits.append(digits_img)
            labels.append(1)
        elif('Two' in img):
            digits.append(digits_img)
            labels.append(2)
        elif('Three' in img):
            digits.append(digits_img)
            labels.append(3)
        elif('Four' in img):
            digits.append(digits_img)
            labels.append(4)
        elif('Five' in img):
            digits.append(digits_img)
            labels.append(5)
        elif('Six' in img):
            digits.append(digits_img)
            labels.append(6)
        elif('Seven' in img):
            digits.append(digits_img)
            labels.append(7)
        elif('Eight' in img):
            digits.append(digits_img)
```

```
            labels.append(8)
        elif('Nine' in img):
            digits.append(digits_img)
            labels.append(9)


    digits = np.array(digits)
    labels = np.array(labels)
    return digits, labels


#initialize svm with default values, that will be used to classify the
numbers
def svmInit(C=1, gamma=0.50625):
  model = cv2.ml.SVM_create()
  model.setGamma(gamma)
  model.setC(C)
  model.setKernel(cv2.ml.SVM_RBF)
  model.setType(cv2.ml.SVM_C_SVC)
  return model

#train svm on training set
def svmTrain(model, samples, responses):
  model.train(samples, cv2.ml.ROW_SAMPLE, responses)
  return model

#runs the svm predictions algorithm
def svmEvaluate(model, samples):
    #predictions = svmPredict(model, samples)
    predictions = model.predict(samples)[1].ravel()
    print("Predictions: " + str(predictions))

    return predictions


#initializes hog_descriptor values, used for running svm
def get_hog() :
    winSize = (28,40)
    blockSize = (16,16)
    blockStride = (4,4)
    cellSize = (16,16)
    nbins = 9
    derivAperture = 1
    winSigma = -1.
    histogramNormType = 0
    L2HysThreshold = 0.2
    gammaCorrection = 1
    nlevels = 64
```

```
    signedGradient = True

    hog =
cv2.HOGDescriptor(winSize,blockSize,blockStride,cellSize,nbins,derivAperture,winSigma,histogramNormType,L2HysThreshold,gammaCorrection,nlevels,
signedGradient)

    return hog
    affine_flags = cv2.WARP_INVERSE_MAP|cv2.INTER_LINEAR


#Does all initialization and setup/training
def runSetup():

    print('Loading digits from digits.png ... ')
    # Load data.
    digits, labels = load_digits('digits.png')



    print('Shuffle data ... ')
    # Shuffle data
    rand = np.random.RandomState(10)
    shuffle = rand.permutation(len(digits))
    digits, labels = digits[shuffle], labels[shuffle]


    print('Defining HoG parameters ...')
    # HoG feature descriptor
    hog = get_hog();

    print('Calculating HoG descriptor for every training image ... ')
    hog_descriptors = []
    for img in digits:
        #print(img.shape)
        hog_descriptors.append(hog.compute(img))
    hog_descriptors = np.squeeze(hog_descriptors)


    digits_train = digits
    labels_train = labels
    hog_descriptors_train = hog_descriptors

    digits_test = None
    labels_test = None


    print('Training SVM model ...')
```

```
    model = svmInit()
    svmTrain(model, hog_descriptors_train, labels_train)

    return model

#pass in numbers to be evaluated by model, returns predicted number results
def evaluateNumbers(model, numbers):

    num_descriptors = []
    hog = get_hog()

    for num in numbers:
        #print(num.shape)
        num = cv2.resize(num,(28,40))

        #cv2.imshow('numbers',num)
        #cv2.waitKey(0)

        num_descriptors.append(hog.compute(num))
    num_descriptors = np.squeeze(num_descriptors)

    print('Evaluating model ... ')
    predictions = svmEvaluate(model, num_descriptors)
    numlist = []
    for p in predictions:
        build = []
        p = int(p)
        numlist.append(p)

    return numlist
```

External Code:

classification.py was originally based on this tutorial teaching how to use smv's in opencv:

https://www.learnopencv.com/handwritten-digits-classification-an-opencv-c-python-tutorial/

Source code: https://github.com/spmallick/learnopencv/tree/master/digits-classification

It has been substantially modified to suit this tasks requirements.

I modified a fairly generic use of bubble sort, in ***numberExtraction.py***, ***process_sign(...)***

To order some contours from left to right, based on the leftmost x coordinate values

https://stackabuse.com/sorting-algorithms-in-python/

find_squares(..) in ***numberExtraction.py*** was originally based on example 3 from this website. It has since been heavily modified.

https://www.programcreek.com/python/example/70440/cv2.findContours

order_points(..) and ***warpRotation(...)*** from ***numberExtraction.py*** were originally based on this tutorial, on how to straighten an angled rectangle and its contents. It has had some slight modifications since.
https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/

I have used some unmentioned snippets from various stack overflow forums as well, but they are unmentioned as they are usually fairly generic implementations of openvcs functions. And do not contain much overlap with my code beyond that.

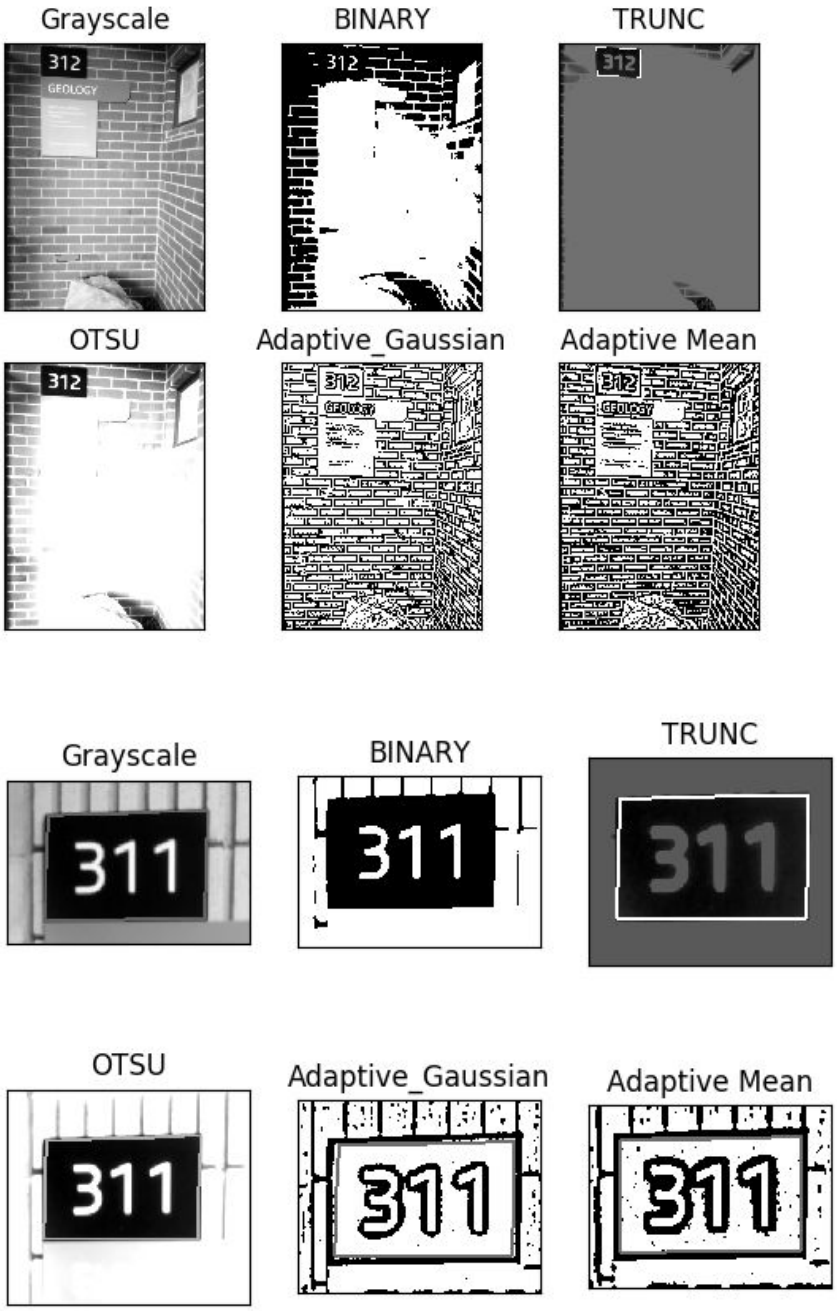
Progress and Process:

Initially on trying to preprocess the images, to isolate the building sign, I just applied some basic filters, like blurs or sharpening and converted to grayscale.

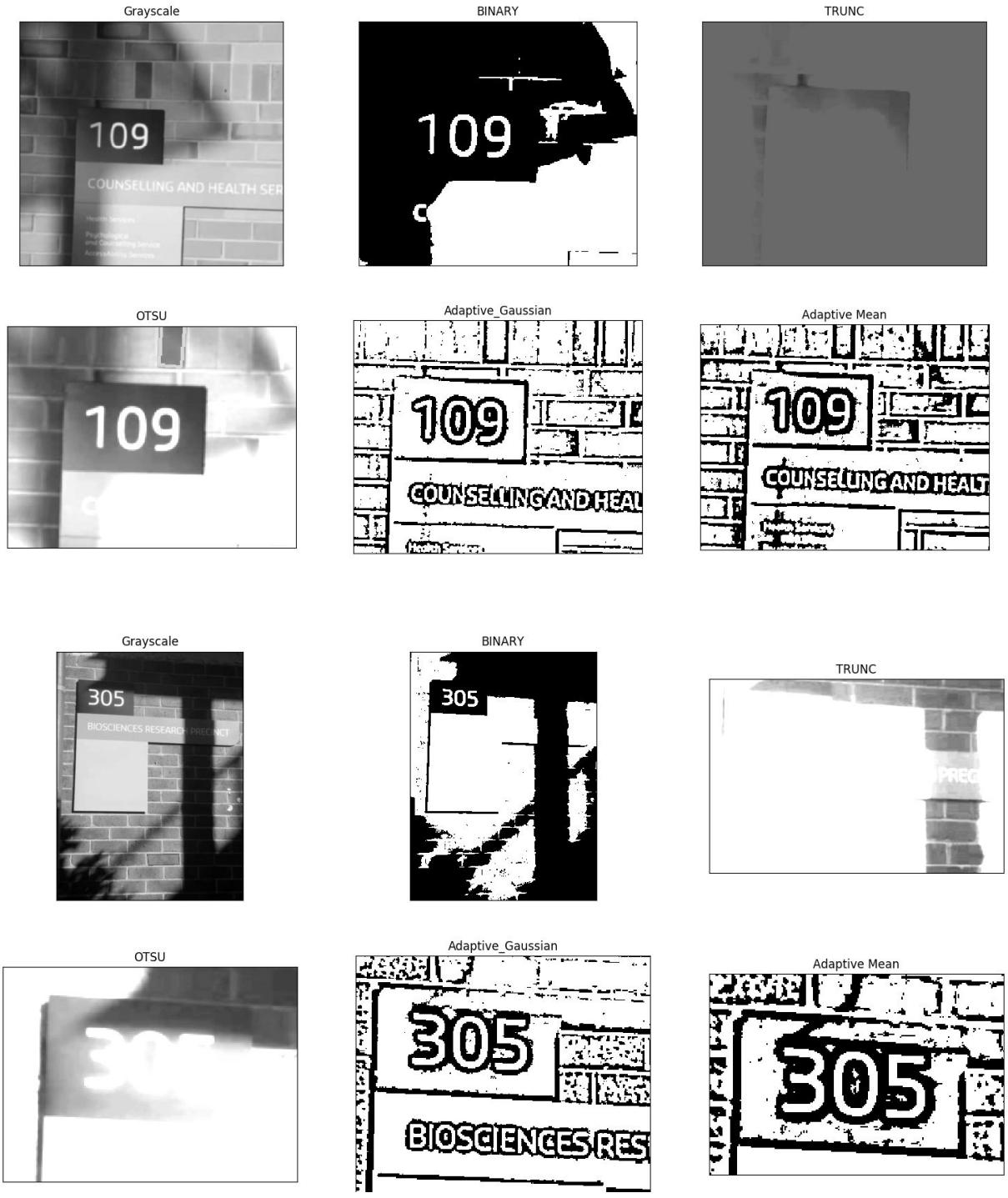
I ran different thresholding methods to see which ones worked best.

And then applied a contour algorithm to find the best rectangle shape.

I had some success with this right away as the sign is black and the text is white, on most images there is a threshold value that will isolate the sign. And you can use adaptive thresholding methods to choose the threshold value.



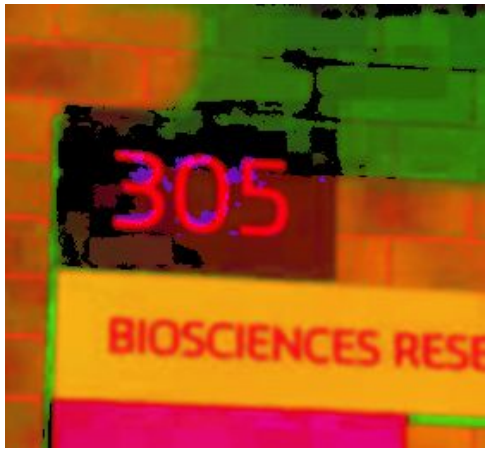
But there were a few pictures where this didn't work, as shadows and light influence the threshold values. So in some pictures the darkest background colours where darker than the brightest black sections. This means there is no thresholding value that can isolate the sign.



To get around this I did some preprocessing using hsv filtering, I converted the image to hsv and filtered based on saturation value. Any substantial colour within the image would be set to (255,255,255) white.

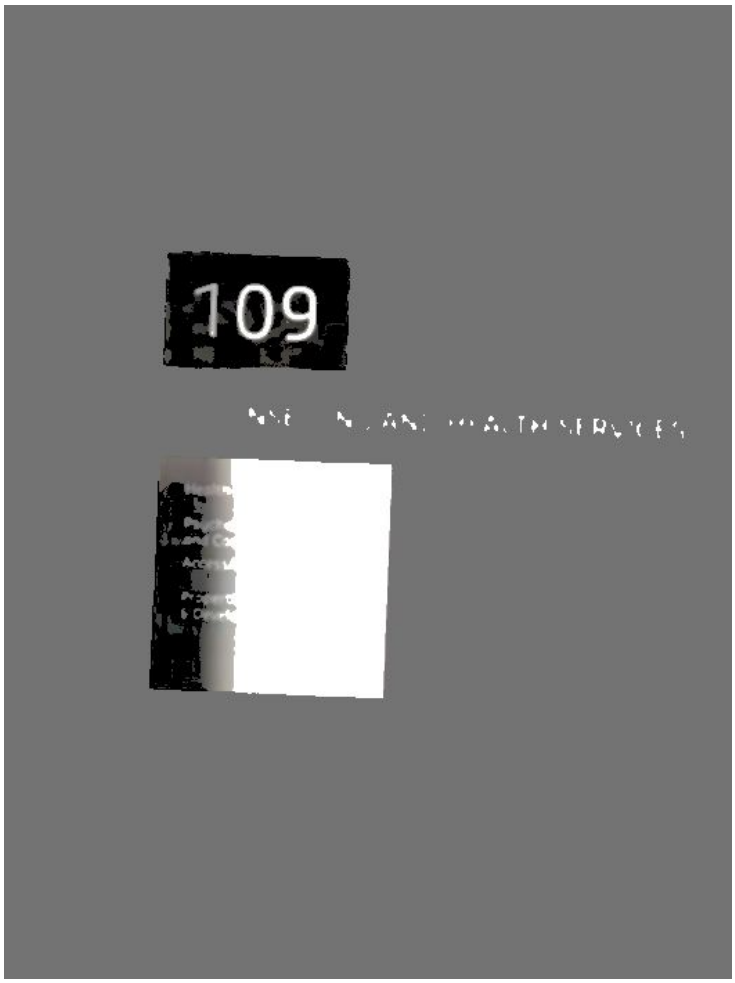
This worked pretty well, except even on the black and white sign, the combination of white and black contains blue values in some images, so I used hsv filtering to darken any dark blue values that fell within this range.

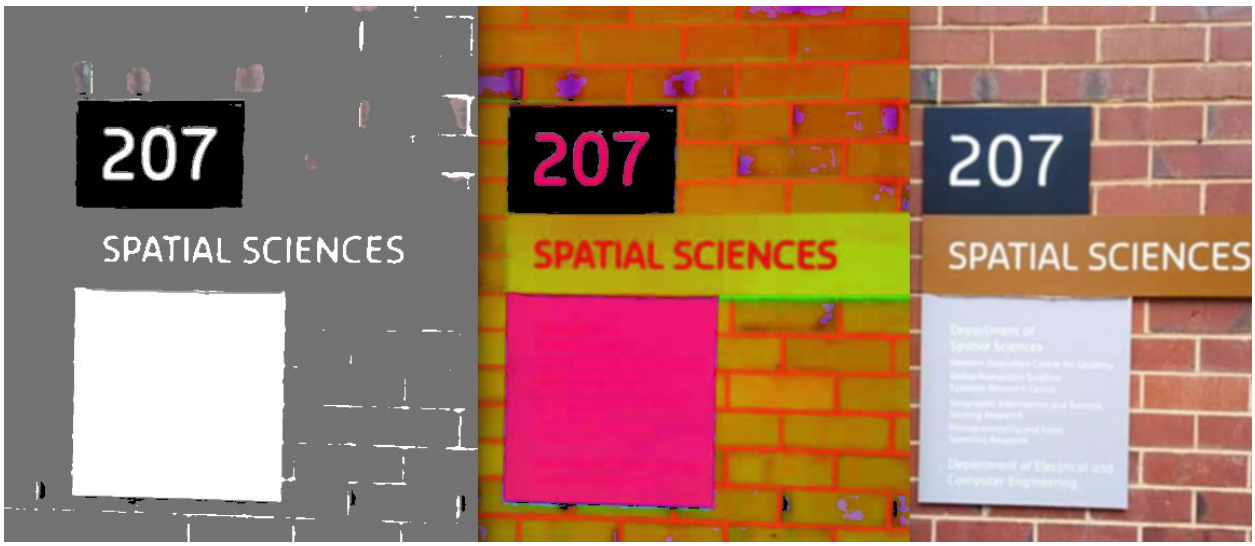
There was also a similar problem with red bricks next to shadows, also making the black signage more red. This was more difficult though as darkening dark black,red values would also blacken large sections of shadowed red brick walls.



The then filtered it so that any other colours above a certain saturation would be set to gray (115)

This worked well and I was able to isolate the box in almost every case.





I did some filtering to remove noise and bad contours, and then rotated the image so the numbers were straight.

I extracted the numbers using bounding boxes ordered by their leftmost x coordinate.

I ran it through a svm to classify the numbers, which used the given training data for the assignment.

Success Rate:

The program worked on all of the testing images except for one. This was due to the dark red brick problem I mentioned earlier, but could definitely be fixed with more time, as the processed image is still pretty solid.

When the box is detected, processing the numbers inside and classifying them worked on all of the test data given.

Task 2:

Unfortunately, I ran out of time to finish my working on task 2 for this submission. I think a lot of what I have done in task one can be transferred over, at least for a moderate success rate. And the number/symbol classifier that has been built will remain nearly identical.

I will probably reupload with task 2 added later.