CURTIN UNIVERSITY
FACULTY OF SCIENCE AND ENGINEERING: SCHOOL OF CIVIL AND MECHANICAL ENGINEERING

Machine Perception Final Assignment

Student name: *Tawana Kwaramba: 19476700*

Course: *Machine Perception - COMP3007* – Lecturer: *Senjian An*
Due date: *October 30, 2020*

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:		Student ID:	
Other name(s):			
Unit name:		Unit ID:	
Lecturer / unit coordinator:		Tutor:	
Date of submission:		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: _____ Date of signature: _____

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

Contents

I Discussion of programme Implementation **1**

1 Overview of programme **2**

2 Approach to image extraction **3**

3 Approach to image classification **6**

II Discussion of produced Results **8**

4 Discussion **11**

5 references **12**

Appendices **13**

A main.py **13**

B Trainer.py **15**

C Image.py **19**

D ImageLoader.py **46**

E Colours.py **51**

F Errors.py **52**

List of Figures

1 Results for the house number 35 9

2 Results for the house number 302 9

3 Results for the house number 71 10

4 Results for the house number 48 10

5 Results for the house number 26 10

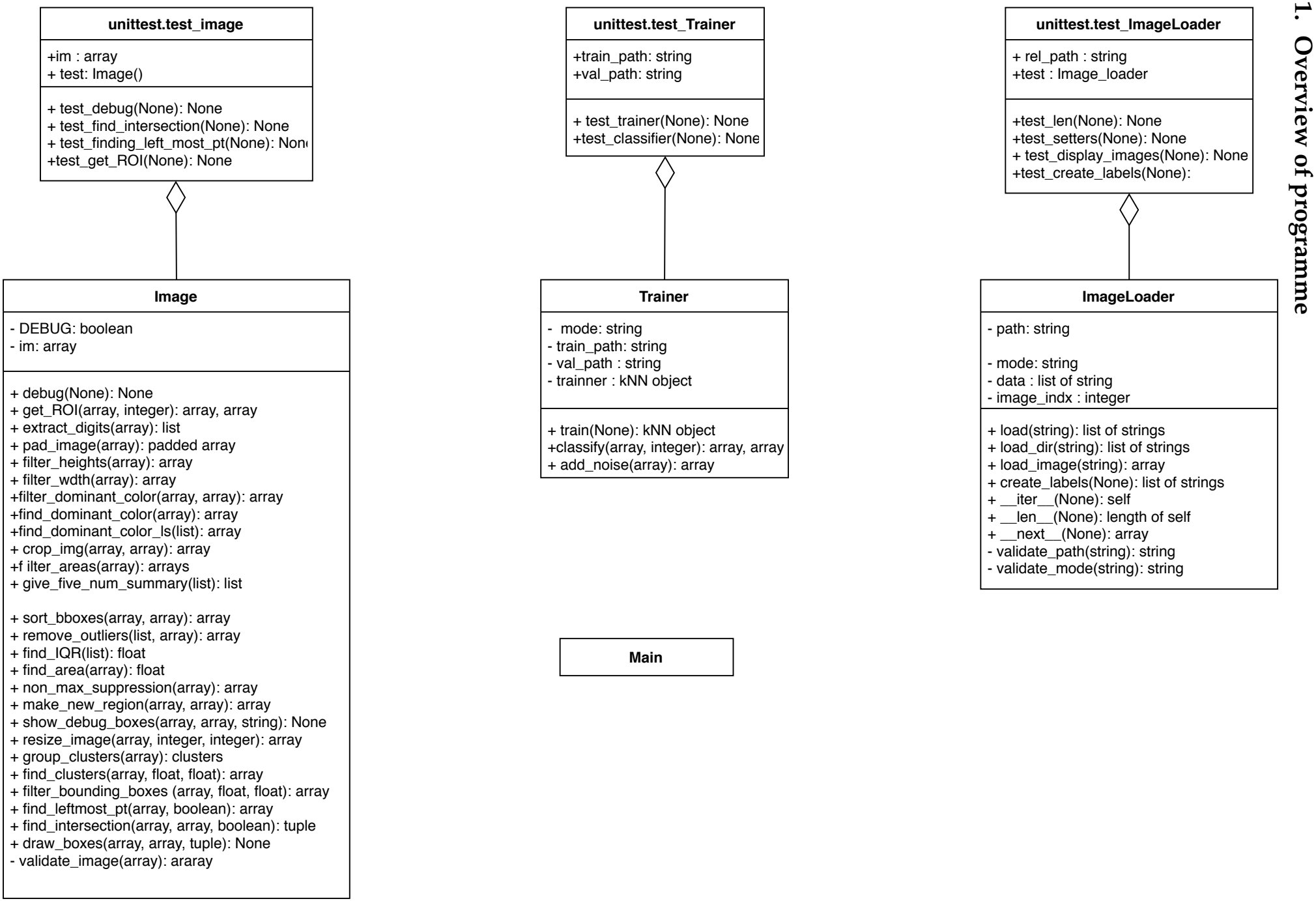
6 Results for the house number 94 11

List of Tables

Part I

Discussion of programme Implementation

1. Overview of programme



2. Approach to image extraction

This report won't discuss the derivation of specific thresholds and numbers, as those numbers were arrived largely from trial and error. Albeit, throughout the report I will be mainly focusing on the ideas which this programme hinges on, and I will be only discussing the pipeline which ultimately gave the best results otherwise, the report will become lengthy. Furthermore, the discussion of handling file input and output won't be discussed as this process is quite trivial. The assignment was approached by building on top of the algorithms and process built in the practicals, and from assignment one additionally from algorithms and information sourced from YouTube videos and online tutorials, the specific algorithms sourced from the tutorials are referenced in the actual programme. /par

Computers are not typically as smart as humans, with humans we detect certain features straight away such as digits hence, for a computer to be able to detect features of interest we have to give the computers certain bounds and limits on how the item of interest can be (reference). Therefore, with this in mind, the basic approach of this programme is for the algorithm to detect as much bounding boxes around the blobs of the image, and to filter out any bounding box which is not a number this process is allegorical to sifting out wheat. /par

The first step in this pipeline is to pre-process the image to enable a blob detection algorithm to detect the blobs efficiently and effectively in the image (reference). The pre-processing of the image includes turning the image into the grey-scale colour space as MSER requires the image to be in this colour space. Additionally, grey-scale colour space decreases the memory requirement of the image while keeping the vital information about the image hence, allowing for faster processing times, and blob detection times (reference). Another pre-processing step which may be required is re-sizing the image if it's above 900x900 pixels, I have discovered through experimenting and the validation images this algorithm will work well for images below 900x900 pixels this is mainly due to the nature of MSER. Additionally, an image above 900x900 will be great memory overhead hence slowing down processing and detection times of the algorithm (reference). Furthermore, the image needs to be blurred, so the memory required for the image is decreased but the important features of the image are kept such as edges and corners of the image, the image in the programme will be blurred through the gaussian kernel (reference).

Thereafter, the image is ready for foreground and background segmentation, and noise removal. This is achieved through the application of Otsu thresholding which will convert the image into a black and white image whereby the black will represent the background of the image, and the white will represent the foreground of the image (reference). This is done because we know that the region of interest (digits) is going to be in the foreground image. Thereafter, the canny edge is conducted on this image, so that algorithm can pick up the edges of the image, this will allow pre-filtering of noisy edges and will pick up the eminent features of the image since gaussian blur and thresholding have already been applied to the image. Once a binary image is created of the

edges of the image, morphological operations are done; the first operation which is conducted is erosion, the purpose of this is to remove the very small edges or spots in the image i.e. to remove noise from the image. After the noise has been removed we then dilate the image the purpose is to make the left interesting features of the image to be a lot fuller, to fill inside the edges of the number, and to remove any noise inside foreground areas. After the pre-processing of the image, the image is ready for blob detection through MSER.

The choice of blob detection was MSER as this was the better-suited blob detection algorithm for the extraction of digits. This is because MSER looks for connected areas, areas of uniform intensity, and an area which is surrounded by a contrasting background (An 2020) which are all criteria which are best suited for detecting digits. It is expect the area of the digits to be connected especially after the morphological dilation operation conducted in the image pre-processing, it's expect the digit found in the image to be of the same colour hence in a grayscale image this will appear as the same intensity, and finally, we're going to expect the digits to be sitting on some form of background or plaque hence the digits will be surrounded by a contrasting background. Therefore, making MSER a feasible algorithm for digit detection given the training data.

The last overall stage of the extraction of digits is applying the filters on all the bounding boxes, and deleting bounding boxes which are not digits hence by the end the remaining bounding boxes should be the desired digits. The first filtering which is conducted is concerning the dimension of the bounding boxes, we know that the digits are going to be tall hence digits will have a longer height than width therefore, we're going to calculate the ratio of height to width of each bounding box and if the ratio is an integer greater than one the height of the bounding box is greater than the width hence, this bounding box is most likely to be a digit. With this observation introduces the first limitation of this algorithm, filtering digits in this manner will not allow detecting digits when the image is 90 degrees the width of the bounding boxes will become the height, and the height will suddenly become the width, therefore, if this algorithm was applied to an image which has been rotated 90 degrees it will filter away the digits in the image. Furthermore, it's expected that this algorithm will slow down linearly with increasing image size as it will access every single bounding box once in the image thus been an O(N) algorithm.

The next filtering which is done is the clustering and the grouping of bounding boxes, we can assume from the given training data that the digits in the test pictures are going to relatively close to one another at least no more than the width of the bounding box of a digit. Therefore, this algorithm will go through each bounding box, and try to find the pair bounding boxes i.e. a bounding box which is close to the current bounding box then the algorithm will make those two boxes a cluster. The manner this algorithm finds clusters can potentially introduce limitations as boxes can be close together but not be the digits hence, this algorithm can introduce more noise into the image if the filtering in the last step was unsuccessful. Furthermore, it's expected that this algorithm will increase exponentially with the size of the image, and the number of the bounding boxes as this algorithm is an $O(N^2)$ as it touches each bounding boxes at least two times when doing comparisons in one sweep hence doing NxN comparisons. After the group of clusters are

found, these pair of bounding boxes are going to be turned into one bounding box hence, this algorithm will look at each pair passed in from finding the bounding boxes, and it will determine the bigger box out of the pair, and will make that bounding box the new bounding box which will represent that pair.

Sequentially, the algorithm will filter the remaining bounding boxes concerning the median area of the remaining bounding boxes, this algorithm is implemented in this order because the dominant remaining bounding boxes are the bounding boxes which are surrounding the digits with extremely small boxes, and extremely large boxes remaining. Hence, since the digits are going to be relatively the same size we will expect that the median area of the bounding boxes to be the area which is a digit or very close to a digit. Therefore, the median area is found, and the interquartile range for the areas of the bounding boxes are found, and any box which lays below or above that inter-quartile range is filtered out. An obvious limitation of this filtering is that it heavily relies on the fact that the previous filters have successfully filtered out a great proportion of the bounding boxes otherwise, this algorithm may filter out the digits if they are still a great proportion of clusters remaining as these clusters can shift the median and filter out the actual digits in the image. This algorithm is expected to linearly increase in time complexity with the increasing number of bounding boxes, and image size as this algorithm has a time complexity of $O(N)$ as this algorithm only touches each element in its list at least once at each comparison.

Thereafter, non-max suppression is applied to the remaining bounding boxes, non-max suppression is the idea of getting smaller boxes which are inside a bigger box, and or deleting boxes which overlap with another box to a certain degree (Sambasivarao 2019)(Rosebrock 2014). Detecting digits such as a 0 and 8 will result in the inner parts of those digits been detected as a bounding box if this isn't filtered out this will impact how the algorithm will create the region of interest as the algorithm is going to join the leftmost point with the rightmost corner of the bounding box which is the furthest to the right side relative to the left corner. Therefore, these little areas are going to be filtered out to avoid the region of interest been from the left most corner to the right side corner of one of these boxes. An evident limitation of this algorithm is the use case where the digits are on a plaque where the height of the plaque is greater than the width, hence this bounding box didn't get filtered out by the previous algorithm and this algorithm will delete each box inside of it which would be the digits for the image. This algorithm is expected to exponentially increase with image size, and with the number of bounding boxes found in an image as the time complexity of this algorithm is going to be $O(N^2)$ as it's going to do N by N comparisons on each sweep of the algorithm.

Afterwards, simple filtering of the width and the height of the bounding boxes is conducted relative to the median positions of the bounding boxes as the filtering done in the previous steps will have isolated the digits in the image. The same manner as the filtering of the areas was conducted in, is the same manner as these two filters will be conducted in, and these algorithms will have similar time complexities and limitations.

Then after, just as an extra pre-cautious step and to ensure that all the noise in the image is completely removed filtering of the bounding boxes is done relative to the dominant colour in each bounding box. It's expected that the bounding box of each digit will have the same dominant colour, and they'll be more bounding boxes with digits left in this stage. Albeit this assumption introduces the limitation of this algorithm which is that it heavily relies on the performance of previous filtering algorithms hence, if the previous algorithms didn't filter out sufficient amount of boxes this algorithm may end up filtering out the wrong type of boxes. After each dominant colour is found, the idea of filtering each bounding box is the same as the idea discussed when filtering for the area in the image.

The last step is to crop the region of interest, and the digits out of the image, by this stage the filters should've filtered out all the bounding boxes which are not digits. Thus, we can extract each of these bounding boxes, and expect to find digits at each position of these bounding boxes. These extractions are the ones which are going to be used when trying to determine what the house number is in the given image. Additionally, from the remaining bounding boxes the algorithm will find the leftmost point in the image, and the rightmost point of the image and join these points together to create a new bounding box, this bounding box will be the bounding box which will contain all the digits of the image.

3. Approach to image classification

The classification of this programme is done through a simple implementation of k nearest neighbour (kNN) even though more sophisticated and better-performing classifiers such as support vector machines (SVM), a plethora of neural networks, template matching, and surfeit supervised classifiers could've been used to classify the digits but weren't used because; k nearest neighbour is highly feasible for this application the digits are going to be on a high contrast background with the digits been a unique colour, other classifiers have extra steps and process for classifying which is not need in this use case, and additionally, the fundamental design principle of this programme is KISS (keeping it simple stupid) therefore using other classifiers will over-complicate the classification process of this algorithm.

Before, we can train the kNN algorithm we need to prepare our data. Typically, you would initially re-size all the training data so that they're the same dimension although this programme doesn't do this step as the provided training data is homogeneous in size. Therefore, all we need to do is turn our multiple dimensioned images in row vectors, and for each image, we should have an accompanying list with a label on what each row vector is supposed to be. Then after the data is prepared, we can make kNN classifier by training the row vectors concerning the labels provided. A prevailing problem with supervised learning algorithms is that they tend to overfit to their training data, therefore, to combat against that the training has some additional noise added to it to avoid over-classification by kNN algorithm.

Thereafter we created our classifier we can use the classifier to classify our desired data. Before we classify our desired data we need to ensure that the data is appropriate for classification hence, we need to ensure that the data to be classified is the same size as the training data which was provided and that the image has some padding from the border of the image and the item to be classified otherwise, this may result in some in-correct classification of the data. Consequently, we use the kNN classifier created in the previous section to classify the data of interest.

Part II

Discussion of produced Results

this image produced a bounding box with the follow coordinates: [8, 9.4, 66, 54]
The classifier of this image produced the following numbers: 33

Figure 1: Results for the house number 35

this image produced a bounding box with the follow coordinates: [198, 147, 600, 103]
The classifier of this image produced the following numbers: 302

Figure 2: Results for the house number 302

this image produced a bounding box with the follow coordinates: [76, 90, 59, 50]
The classifier of this image produced the following numbers: 70

Figure 3: Results for the house number 71

this image produced a bounding box with the follow coordinates: [48, 1, 210, 145]
The classifier of this image produced the following numbers: 4826

Figure 4: Results for the house number 48

this image produced a bounding box with the follow coordinates: [80, 90, 107, 79]
The classifier of this image produced the following numbers: 28

Figure 5: Results for the house number 26

this image produced a bounding box with the follow coordinates: [56, 54, 109, 103]
The classifier of this image produced the following numbers: 94

Figure 6: Results for the house number 94

4. Discussion

As can be seen from the produced results, the image classifier is unable to classify some images successfully as seen figure 1, 3, and 4. Since the classifier can only classify half of the images successfully it can be said that the classifier has a classification accuracy of 50%.

They're two reasons for the low classification score of the images firstly, some of the images weren't segmented properly i.e. the bounding boxes were bounding more than just the digits namely the results produced in figure 4. Therefore when the image class segmented the bounding boxes out of the image it also segmented the brick which was on the top right of the image hence, the kNN classify will try to fit a number to this segment. Since they are no digit in this segment it will output random numbers. This behaviour is observed through the outputted number of the classify of 4826, the first two digits are the correct classified digits of the house number and the 26 is what kNN thinks the brick is. Additionally, for this case, the results could've been improved by running a check of L2-Norm distance calculated for the segmented image and if that distance isn't above a specific threshold the classify won't classify that digit as it doesn't closely represent any number in its data set. But the nature of kNN is that it will find the closest fitting distance for an image regardless of how far that distance is away from any data point in the dataset.

Secondly, poor classification can be explained by the training data produced. Some digits in the training data set have more examples than other digits this will cause the kNN classifier to overfit to digits with more examples in the training data set namely, in the training data set they were 10 examples of a 1, and 13 examples of 0 hence, kNN will more classify something which closely represents a 0 as zero than the digit it is as can be seen in figure 3. Moreover, they're 10 examples of a 5, and 18 examples of a 3 hence the classify will classify anything which marginally looks like a 5 as a 3 as seen in the results in 1. For future work to produce better results, a function can be implemented which gets the minimum number of training examples from each labelled data-set and truncates all the data sets to be the length of the minimum data set found so the kNN classifier won't over-fit to any specific digit in the training data set. Furthermore, since kNN segments images based on colour, a common approach is to introduce noise to each example in the data set this approach did provide better classification performance. In the earlier stages of

the algorithm whereby noise wasn't introduced to the data set, the programme would classify the four seen in figure 6.

5. references

An. Senjian. 2020. "Machine Perception Lecture 03". Power point slides..https://learn-ap-southeast-2-prod-fleet01-xythos.s3.ap-southeast-2.amazonaws.com/5dc3e34515a0e/4348643?response-cache-control=private%2C%20max-age%3D21600&response-content-disposition=inline%3B%20filename
8%27%27lecture03_feature_detection.pdf&response-content-type=application%2Fpdf&X-Amz-Algorithm
HMAC-SHA256&X-Amz-Date=20201030T210000Z&X-Amz-SignedHeaders=host&X-Amz-Expires=2160
Amz-Credential=AKIAYDKQORRYZBCCQFY5%2F20201030%2Fap-southeast-2%2Fs3%2Faws4_request
Amz-Signature=4263cc0edf55555ee8cd4010f52d0ea712914f2cc1fbf608a718041516f0a45c

Rosebrock, Adrian. 2014. "Non-maximum Suppression for object detection in Python". Pyimage-search. https://www.pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/

Sambasivarao, K. 2019. "Non-maximum Suppression (NMS)". https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c

Appendices

A. main.py

```
"""
FILENAME: main.py

AUTHOR: Tawana Kwaramba: 19476700
LAST EDITED:

PURPOSE OF FILE: this is the main of the programme hence, it is to facilate
the functionaility of this assigment. Therefore, it brings all classes create
in order to detect numbers given an input image, crop the digits, save the
necessary files, and to classfity the images
"""
import argparse
from Trainer import *
from Image import *
import os
from ImageLoader import *
from Colours import *
#paths of the located files:
test_path = '/home/student/test/'
trainning_path = '/home/student/train/'
val_path = '/home/student/val'

if __name__ == '__main__':
    #extracting the region of interest to the output file
    trainning_images = Image_Loader(trainning_path, 'BGR')
    test_images = Image_Loader(test_path, 'BGR')
    trainner = Trainer(train_path=trainning_path, val_path=test_path,
            mode='BGR')

    results = []
    dists = []
    digits_ls = []
    #im_id is needed so that we can save the files with a unique id but with
    #the same starting string
    for im_id, image in enumerate(test_images):
        try:
            digits = Image(image, im_id)
```

```
#this  is bad programming practice. Although, they is far too many
#things which can go wrong in terms with the assertions and exception
#thrown by openCV, and the bounding boxes. Therefore, for efficient u
#of time, I am going to catch all of them
except:
    #if that image throws an exception, it means that the bounding bo
    #of that image can't be found and it has failed the extraction
    print(red+"image couldn't extraxt images as"+
            " bounding boxes couldn't be found"+reset)


#indexing 1 as I only the result of the classifify as classify return
#the result and the distance of the image
result, other = trainner.classify(digits.im[1])
base_file_name = 'output/House'
#creating the file name based on the numbers found
#I need to use list operations in able to convert the number found in
#numpy into a whole string
house_num = result.tolist()
#the results of the numbers are stored in a 2-dimensional array, wher
#they's only  one number inside in the second dimension of the array
#hence, I am just extracting all those numbers from that second
#dimension
house_num = [int(house_num[ii][0]) for ii in range(len(house_num))]
#taking a house number from an array of strings, and converting it to
#just a string by itself
house_num = ''.join(map(str, house_num))
print(green+"HOUSE NUMBER:"+reset, house_num)
complete_file_name = base_file_name + house_num + ".txt"

with open(complete_file_name, 'w') as inStrm:
    inStrm.write('Building {}'. format(house_num))
```

B. Trainer.py

```
"""
AUTHOR: Tawana Kwaramba: 19476700
LAST EDITED:

PURPOSE OF FILE: the purpose of the file is to facilate the trainning of the
digits for this programme, and it's also to take in some in-coming digits and
classify those digits in relation to the trainned data. This file utilise
kNN for trainning and classification of its data
"""
from Image import *
from ImageLoader import *
from Colours import *
import numpy as np
import pickle

#This class is meant to be an abstract class but, #I have choosen to make thi
#its own object so I can test this class by iself, and when the individual
#trainers inherent from this, I can just test the train method for that speci
#trainning method

class Trainer(object):
    def __init__(self, **kwargs):
        self._mode = kwargs['mode']
        self._train_path = kwargs['train_path']
        #I am going to use the validation path as the same as the
        #test path for this data as they're doing the same thing
        self._val_path = kwargs['val_path']
        self._trainner = self.train()

    #==========================ACCESORS=====================================
    @property
    def test_set(self):
        return self._test_set

    @property
    def train_set(self):
        return self._train_set

    @property
    def val_set(self):
```

```
        return self._val_set

    @property
    def trainner(self):
        return self._trainner


    #=========================PUBLIC METHODS=============================
    def train(self):
        """
        import:None
        Export: knn (kNN clusters objext)

        PURPSOSE: it is to train the kNearest neigbour classsify given the
        trainning data of digits from 0 to 9

        this algorihtm is adapted from:
            Pysource. 2018. "knn handwrittend digits recoginition – OpenCV 3.
            with python 4 Tutorial 36. https://www.youtube.com/watch?v=tOVwVv
            _Pg&ab_channel=Pysource
        """
        trainning_file_name = "kNN_classfier"
        labels_file_name = "kNN_labels"

        #checking if a trainning file already exists in the current directory
        #if it does, load that file

        if trainning_file_name in os.listdir() and \
        labels_file_name in os.listdir()  :
            print(green+"reading␣in␣serilised␣file...."+reset)
            #load the file if it exists, this will allow for faster
            #classification times if the module has been already pre-trainned
            with open(trainning_file_name, 'rb') as inStrm:
                trainning_data = pickle.load(inStrm)

            with open(labels_file_name, 'rb') as inStrm:
                labels_data = pickle.load(inStrm)
        else:
            print(green+"creating␣a␣new␣file...."+reset)
            #create a new file
            in_path = self._train_path
            #using image loading object for faster and efficient image loadin
```

```
            trainning_im = Image_Loader(in_path, self._mode)
            #creating the labels based on the file name which the number belo
            #too
            labels = trainning_im.create_labels()

            trainning_data = []
            labels_data =[]

            #pre-pranning out trainning images, and our trainning data
            for label in labels:
                #opening each digit file and storing it at the trainning_im
                #object
                trainning_im.path = in_path +str(label)
                #accessing every image inside the trainning_im object
                for im in trainning_im:
                    im = self.add_noise(im)
                    im = Image.pad_image(self, im)
                    im = Image.resize_image(self, im, 28, 40)
                    trainning_data.append(im.flatten())
                    labels_data.append(label)
            trainning_data = np.array(trainning_data, dtype=np.float32)
            labels_data = np.array(labels_data, dtype=np.float32)

            with open(trainning_file_name, 'wb') as inStrm:
                #creating a serilised file for future use
                pickle.dump(trainning_data, inStrm)

            with open(labels_file_name, 'wb') as inStrm:
                #writing a labels serilised file for future use
                pickle.dump(labels_data, inStrm)

        knn = cv.ml.KNearest_create()
        knn.train(trainning_data,cv.ml.ROW_SAMPLE, labels_data)

        return knn

    def classify(self, images, k=8):
        """
        IMPORT: images (list of uint8 numpy arrays i.e. images)
        EXPORT: results (string): the label  which the classify to the images
                dist (numpy array): contains the L2 norm distance of each
                result found
```

```
    PURPOSE: it's to assign a label to inputted images

    this algorihtm is adapted from:
        Pysource. 2018. "knn handwrittend digits recoginition – OpenCV 3.4
        with python 4 Tutorial 36. https://www.youtube.com/watch?v=tOVwVv
        _Pg&ab_channel=Pysource
    """
    test_data = []
    for im in images:
        #we need to pad the image, so the area of interest is away
        #from the border of the image
        im = Image.pad_image(self, im)
        #this needs to be the same size as the provided trainning data
        im = Image.resize_image(self,im, 28, 40)
        test_data.append(im.flatten())
    #knn classifier only accpets numpy arrays
    test_data = np.array(test_data, dtype=np.float32)
    ret, result, neigbours, dist = self.trainner.findNearest(test_data, k
    return result, dist



#AUGMENTATION OPERATION METHODS
def add_noise(self, im):
    """
    IMPORT: im (numpy array data type: uint8)
    EXPORT: im (numpy array data type: uint8)

    PURPOSE: it's to add noise to an image, to stop the trainner to
    over-fitting to the provided trainning data.
    """
    mean = (5,5,5)
    sigma = (5, 5, 5)
    noise = np.zeros(im.shape, dtype='int8')
    #75 works well
    cv.randn(noise, (25, 25, 25), (200, 200, 200))

    ret = cv.add(im, noise, dtype=cv.CV_8UC3)
    return ret
```

C. Image.py

```
"""
FILENAME:   Image.py

AUTHOR: Tawana Kwaramba: 19476700
LAST EDITED:

PURPOSE OF FILE: the purpose of this file is to facilate any based image
operations. I.e. segmeneting the image to get the region of interest, and
extracting the digits from the region of interest
"""


from abc import abstractmethod
import numpy as np
from Errors import *
from Colours import *
import cv2 as cv
from statistics import mode

class Image(object):
    def __init__(self, im, img_id):
        #set this to true, if you want to see each step of the image
        #segmentation process
        self._DEBUG = False
        #self._DEBUG = False
        self._im = self.get_ROI(im, img_id)

    #=========================ACCESORS=====================================
    @property
    def im(self):
        return self._im

    @property
    def DEBUG(self):
        return self._DEBUG

    @im.setter
    def im(self, in_im):
        self._im = self.get_ROI(in_im)

    #=========================METHODS======================================
```

```
def debug(self):
    """
    IMPORT: none
    EXPORT: none

    PURPOSE: to act as a toggle to witch the debugging features for this
    class
    """
    if self.DEBUG:
        self._DEBUG = False
    else:
        self._DEBUG = True

#if digits is true, the algorithm will run this function again to extract
#the digits out of the cropped image otherwise, it won't invoke that
#inner function
def get_ROI(self, im, img_id):
    """
    IMPORT:
        im : numpy array of data type uint8
        img_id : integer

    EXPORT:
        cropped_image : numpy array of data type uint8
        digits : numpy array of data type uint8

    PURPOSE: it's to extract the region of interest which is the area whi
    contains all the house numbers in the image, and to extract each digi
    inside that cropped area
    """

    #normilising the images so the colors in the image can be
    #consistent
    norm_img = np.zeros(im.shape[:2])
    im = cv.normalize(im, norm_img, 0, 255, cv.NORM_MINMAX)

    #correcting the colors of the image so over exposure in light
    #doesn't alter the results obtained

    #20 worked really well for this image
    im  = cv.convertScaleAbs(im, alpha=1, beta=-10)
```

```
#to determine if we need to re-adjust our bounding boxes to meet the
#sepcifications of the original images
resized = False
if im.shape[0] > 900 and im.shape[1] > 900:
    resized = True
    im = self.resize_image(im, 536, 884)

#makind sure that we have actually passed in an image, and not anythi
#else
im = self._validate_image(im)
gray = cv.cvtColor(im, cv.COLOR_BGR2GRAY)
#decreasing the required memory the image needs but still keeping the
#important features of the image
gray = cv.GaussianBlur(gray, (5,5), 0)
#thresholding, so we can extract the background and the foreground of
#the image
thresh = cv.threshold(gray, 0, 255, cv.THRESH_BINARY+cv.THRESH_OTSU)[

edge_thresh = 100

#the openCV doc recommends that you will have your upper thresh hold
#twice as the lower threshold

#this doesn't really have any performance boost to be honest
canny_trans = cv.Canny(thresh, edge_thresh, edge_thresh * 2)
cv.waitKey()
#getting the shape and size of the structual element suitable to this
#image. Hence, the window which is going over this image
rect_kern = cv.getStructuringElement(cv.MORPH_RECT, (5,5))

#removing the boundaries of the foreground of the image, so we can
#have less noise around teh actual digits
canny_trans = cv.erode(thresh, None, iterations=1)
#the numbers are a lot smaller at this point hence, we will expand
#those eroded boundaries in order to fill in holes, and to make the
#number fuller
canny_trans = cv.dilate(thresh, None, iterations=1)

if self._DEBUG:
    cv.imshow("found edges after morphology" , canny_trans)
    cv.waitKey()
    cv.destroyAllWindows()
```

```
mser = cv.MSER_create()
regions, bboxes = mser.detectRegions(canny_trans)

if self._DEBUG:
    self.show_debug_boxes(bboxes, im, "original␣bounding␣boxes␣found"

#filtering the bounding boxes relative to the height and width. The
#heights should be greater thna the heights
bboxes = self.filter_bounding_boxes(bboxes)

if self._DEBUG:
    self.show_debug_boxes(bboxes, im, "filtered␣bounding␣boxes")

#the numbers should be relative close to each other hence, we're goin
#to filter out the boxes which are not close to each other
bboxes = self.find_clusters(bboxes, 1.10, 0.25)
#joinning those boxes which are close together, to make one section
bboxes = self.group_clusters(bboxes)

if self._DEBUG:
    self.show_debug_boxes(bboxes, im, "groups␣of␣bounding␣boxes␣found

#by this stage it will just be the numbers left with some noise
#hence we're going to filter out the areas which don't align with the
#numbers in the image
bboxes = self.filter_areas(bboxes)

if self._DEBUG:
    self.show_debug_boxes(bboxes, im, "filtering␣by␣the␣area")


#when we have a zero or an eight. MSER will detect as bounding boxes
#the reigons inside these digits. Hence, we need to remove those regi
#so we can crop the full number successfully
bboxes = self.non_max_suppression(bboxes)

if self._DEBUG:
    self.show_debug_boxes(bboxes, im, "non␣max␣suppression")


#the numbers should be at the relative same heights hence, remove any
```

```
#box which doesn't agree with this height
bboxes  = self.filter_heights(bboxes)

if self._DEBUG:
    self.show_debug_boxes(bboxes, im, "filtering␣by␣the␣heights␣in␣im

#the numbers should be at the relative same widths hence, remove any
#which doesn't agree with this
bboxes  = self.filter_width(bboxes)

if self._DEBUG:
    self.show_debug_boxes(bboxes, im, "filtering␣by␣widths␣of␣the␣ima


#by this point they're still some noise boxes left although, they're
#more boxes which contain the number left in the image hence, we can
#filter these boxes out given the dominant color
bboxes = self.filter_dominant_color(im.copy(), bboxes)

if self._DEBUG:
    self.show_debug_boxes(bboxes, im, "filtering␣done␣by␣dominant␣col


#getting the points required to create our region of interest
#getting the left most - upper most bounding box point
left_pt = self.find_leftmost_pt(bboxes)
#getting the right most lower most bounding box in the image
right_pt = self.find_leftmost_pt(bboxes, True)


#creating a new bounding box which will represent the region of inter
new_region = self.make_new_region(left_pt, right_pt)
cropped_image = self.crop_img(im.copy(),new_region)
#a padding is needed for better digit detection. If the digit is a
#part of the border in some cases that part won't be detected by
#MSER
cropped_image = self.pad_image(cropped_image)

digits = self.extract_digits(cropped_image)

file_name = 'output/DetectedArea' + str(img_id) + ".jpg"
bbox_file_name = 'output/BoundingBox' +str(img_id) + ".txt"
```

```
        #cv.imwrite(file_name, cropped_image)
        self.draw_boxes(bboxes, im, (255,0,0))
        cv.imwrite(file_name, im)
        np.savetxt(bbox_file_name, [new_region], delimiter=',')

        if self._DEBUG:
            self.show_debug_boxes([new_region], im, "new region found")

        if self._DEBUG:
            cv.imshow("extracted area", cropped_image)
            cv.waitKey()
            cv.destroyAllWindows()

        if self._DEBUG:
            for  index, im in enumerate(digits):
                cv.imshow("digit %s" % index, im)
            cv.waitKey()
            cv.destroyAllWindows()

        return cropped_image, digits

    def extract_digits(self, im):
        """
        IMPORT:
                im : numpy array array datatype of uint8
        EXPORT:
            list of numpy arrays of datatypes of uint8

        PURPOSE: is to get the region of interest produced by the image, and
        to extract the individual digits out of this image
        """

        im = self._validate_image(im)
        gray = cv.cvtColor(im, cv.COLOR_BGR2GRAY)
        #decreasing the required memory the image needs but still keeping the
        #important features of the image
        gray = cv.GaussianBlur(gray, (5,5), 0)
        #thresholding, so we can extract the background and the foreground of
        #the image
        thresh = cv.threshold(gray, 0, 255, cv.THRESH_BINARY+cv.THRESH_OTSU)[
```

```
        edge_thresh = 100
        #the openCV doc recommends that you will have your upper thresh hold
        #twice as the lower threshold
        canny_trans = cv.Canny(thresh, edge_thresh, edge_thresh * 2)
        rect_kern = cv.getStructuringElement(cv.MORPH_RECT, (5,5))
        canny_trans = cv.erode(thresh, None, iterations=1)
        canny_trans = cv.dilate(thresh, None, iterations=1)
        #canny_trans_invert = canny_trans.max() - canny_trans

        if self._DEBUG:
            cv.imshow("found edges after morphology" , canny_trans)
            #cv.imshow("the inversion of that image", canny_trans_invert)
            cv.waitKey()
            cv.destroyAllWindows()

        mser = cv.MSER_create()
        regions, bboxes = mser.detectRegions(canny_trans)
        #trying to filter the bounding boxes in relation to the heights, and

        if self._DEBUG:
            self.show_debug_boxes(bboxes, im, "original bounding boxes found"

        bboxes = self.filter_bounding_boxes(bboxes, 1.1, 4.8)
        bboxes = self.non_max_suppression(bboxes)
        bboxes = self.non_max_suppression(bboxes)
        #sorting the bounding boxes so they read left to right, and it's
        #displayed in the right order
        bboxes = sorted(bboxes, key=lambda x: x[0])
        #removing all duplicate bounding boxes in the same row
        bboxes = np.unique(bboxes, axis=0)

        return [self.crop_img(im.copy(), box) for box in bboxes]

    def pad_image(self, im):
        """
        IMPORT: im : numpy array of datatype unit8
        EXPORT: padded image: numpy array of datatype unit8

        PURPOSE: it's to place a black padding around an image, so that
        the numbers of the image don't become a part of the border
        """
```

```
        #number of pixels which we want to pad the image with all around
        row_pad = 3
        col_pad = 3
        npad = ((row_pad, col_pad), (row_pad, col_pad), (0,0))
        return np.pad(im, pad_width=npad, mode='constant', constant_values=0)

    def filter_heights(self, bboxes):
        """
        IMPORT: bboxes : numpy of array of dtype int32
        EXPORT: bboxes numpy of array of dtype int32

        PURPOSE: to filter out the bounding boxes by height so the bounding
        boxes which are relativily around the same height will remain in
        the image
        """
        #sorting the bounding boxes in relation to the poistion on the image.
        #placing the upper most box first in the bboxes list and the lowest
        #box last in the list
        bboxes = sorted(bboxes, key=lambda y: y[1])
        #just creating a list which only contains the heights of the bounding
        #boxes
        heights = [box[1] for box in bboxes]
        #finding the median height of the bounding boxes
        common_height =  np.median(heights)

        #grabbing the bounding box with the lowest height in the image
        #and grabbing its width
        TOL = bboxes[-1][3]

        for indx, box in enumerate(bboxes):
            #a safe gaurd to make sure that we don't try to access an invalid
            #index of the bounding boxes
            if indx < len(bboxes):
                #remove the bounding boxes which are not about the
                #same height as the median value of the bounding boxes
                if (abs(box[1] - common_height)) >= TOL:
                    #you don't want to delete the elements as yet, as
                    #that will make the array smaller, and will cause
                    #python to try to access an index which is out of range
                    bboxes[indx] = [-1,-1,-1,-1]

        bboxes = self.remove_invalid(bboxes)
```

```
        return bboxes

    def filter_width(self, bboxes):
        """
        IMPORT: bboxes  : numpy array of datatype int32
        EXPORT: bboxes  : numpy array of datatype int32

        PURPOSE: the purpose is to filter out boxes which are not
        relatively close to each other in the provided bounding boxes as
        these points are most likely going to be noise in the image
        """
        #sorting the bounding boxes so we can access the boxes from
        #left to the right side of the image
        bboxes = sorted(bboxes, key=lambda x: x[0])
        #grabbing just the widths of all the bounding boxes in the image
        widths = [box[0] for box in bboxes]

        #the numbers in the image should be the middle number by this stage
        common_width = np.median(widths)

        #grabbing the right most box
        TOL = bboxes[-1][2] * 4

        for indx, box in enumerate(bboxes):
            #a safe gaurd to ensure the algorithm doesn't try to index
            #outside of the list
            if indx < len(bboxes):
                if (abs(box[0] - common_width)) >= TOL:
                    #you don't want to delete the elements as yet, as
                    #that will make the array smaller, and will cause
                    #python to try to access an index which is out of range
                    bboxes[indx] = [-1,-1,-1,-1]


        bboxes = self.remove_invalid(bboxes)

        return bboxes


    def filter_dominant_color(self, img, bboxes):
        """
```

```
IMPORT:
        img : numpy array of dataype uint8
        bboxes: numpy array of datatype int32

EXPORT: bboxes : numy array of dataype int32

PURPOSE: the idea is that the numbers should be on the same
coloured background and the numbers should be the same color aswell
and the dominant color should be either the background or
the foreground of the image. Hence, filter away any box which
are not the same color as the numbers
"""
#normalising the image, removing any effects from brightness and
# contrast
img = cv.normalize(img, img, 0, 255, cv.NORM_MINMAX)
#best color space to use from experimenting
img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
img_sections = []

for box in bboxes:
    #creating a list of images of wahtever is inside the bounding
    #boxes found in the image
    img_sections.append(self.crop_img(img.copy(), box))

section_colors = []
for section in img_sections:
    #finding the dominant color in each of those cropped sections
    #before
    section_colors.append(self.find_dominant_color(section))

#if they is any invalid data remove it from the section colors
#list
section_colors = self.remove_invalid(section_colors)
#go throught the list of dominant colors and find the most
#occurring color in that given list
dominant_color = self.find_dominant_color_ls(section_colors)

#this gave the best results  so far from trial and error of multiple
#values
TOL = [25, 25, 25]

#anything which doesn't have this dominant color should be deleted in
```

```
        for indx, color in enumerate(section_colors):
            #safe gaurd to try stop the algorithm from accessing an
            #invalid index
            if  indx < len(section_colors):
                #determing if all the elments of the color array is
                #realtivley near the domiant color of the image
                if (abs(color - dominant_color) > TOL).all():
                    #deleting the bounding box which doesn't have the
                     #dominant color in it
                    bboxes[indx] = [-1,-1,-1,-1]

        bboxes = self.remove_invalid(bboxes)

        return bboxes

    def find_dominant_color(self, img):
        """
        IMPORT: img : numpy array of dataype unit8
        EXPORT:the  palette : numpy array which will represent the most
        appearing color in the array

        PURPOSE: the purpose is to determing the most appearing color
        in an image i.e. the dominant color of that image
        """
        pixels = np.float32(img.flatten())
        #the should be atleast two dominant colors in an image, the
        #color which the number is, and the background where the number is
        #sitting on in the image
        n_colors = 2

        criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 200, 1)
        flags = cv.KMEANS_PP_CENTERS
        labels, palette  = cv.kmeans(pixels, n_colors, None, criteria, 10,
                flags)[1:]

        counts = np.unique(labels, return_counts=True)[1]

        return palette[np.argmax(counts)]

    def find_dominant_color_ls(self, color_ls):
        """
        IMPORT: color_ls (a list of numpy arrays )
```

```
        EXPORT: dominant_color (a numpy array)

        PURPOSE: to find the domiannt color given a list. This algorihtm is
        going to use brute force to find the dominant color, and therefore
        is going to be very in-efficient to use
        """

        temp_color = None
        #prvious amount of times a previous color has been found
        prev_found_times = 0
        #the number of times the curren color has been found
        found_times = 0

        for color in color_ls:
            temp_color = color
            for count in color_ls:
                #seeing how many times that specific color which we're
                #on is found in the array
                if (count == temp_color).all():
                    found_times += 1

            #this should really be > but aye it works better this way...
            if  found_times >= prev_found_times:
                #setting this to the new dominant color
                dominant_color = color
                prev_found_times = found_times
                #resetting the counter back to 0
                found_times = 0

        return dominant_color


    def crop_img(self, img, bbox):
        """
        IMPORT:
            img : a numpy array of datatype uint8
            bbox : a numpy array of datayype int32

        EXPORT:
            img : a numpy array of datatype uint8

        PURPOSE: is to crop an image in relative to a given bounding box
```

```
        """
        x_l = int(bbox[0])
        y_l = int(bbox[1])
        x_r = int(bbox[0] + bbox[2])
        y_r = int(bbox[1] + bbox[3])

        return img[y_l:y_r, x_l:x_r]


    def filter_areas(self, bboxes):
        """
        IMPORT: bboxes : a numpy array of datatype int32
        EXPORT: bboxes : a numpy array of datatype int32

        PURPOSE: it's to find the median area of the bounding boxes
        and filter any bounding boxes which are not in a range of that
        median of the image. This a clean up algorithm so most of the noise
        boxes have been filtered by only the numbers and a couple of noisy
        boxes will remain in the image
        """
        all_areas = []
        #getting all the areas of each bounding box in the image, so I can
        #fileter out the boxes which are most likely going to be outliers
        for box in bboxes:
            #calculating all the areas of the bounding boxes and adding
            #them to a list
            all_areas.append(self.find_area(box))
        bboxes = self.remove_outliers(all_areas, bboxes)

        return bboxes

    def get_five_num_summary(self, area_ls):
        """
        IMPORT: area_ls (a list of real numbers)
        EXPORT: summary (a list of 5 real numbers)

        PURPOSE: it's to find the statistics of given numbers. Hence, it
        will find the max data point, the min data point, the 25th
        percentile the median, and the 75th percentile data points
        """
        min_area, max_area = min(area_ls), max(area_ls)
        #finding the 1st, 2nd, and 3rd quartile of the given data
```

```
        percentiles = np.percentile(area_ls, [25, 50, 75])
        summary = [min_area, percentiles[0], percentiles[1], percentiles[2],
                max_area]
        #return min_area, max_area, median, average
        return summary

    def sort_bboxes(self, sorted_indxs, bboxes):
        """
        IMPORT:
            sorted_indxs : a list of integers
            bboxes : numpy array of datatype int32

        EXPORT:
            bboxes: numpy array of datatype int32

        PURPOSE: is to sort a numpy array relative to a given list of
        indexs
        """
        temp_bboxes = bboxes.copy()
        for indx, box in enumerate(temp_bboxes):
            #placing everything in their respective poistion in bounding
            #boxes
            bboxes[indx] = temp_bboxes[sorted_indxs[indx]]

        return bboxes

    def remove_outliers(self, area_ls, bboxes):
        """
        IMPORT:
            area_ls : list of integers
            bboxes : a numpy array of datatype int32

        EXPORT:
            bboxes : a numpy array of datatype int32

        PURPOSE: an outlier is a point which will lay siginificantly far
        away from the median value of the data
        """
        area_ls = np.array(area_ls)
        #sorting the area relative to the indexes. Hence, this will
        #only have the sorted indexes of the array
        sorted_indxs = area_ls.argsort()
```

```
#sorting the bounding boxes, so the bounding boxes are put in
#the index which corresponds with its area
bboxes = self.sort_bboxes(sorted_indxs, bboxes)

#actually puting the areas in their sorted order
area_ls = area_ls[sorted_indxs]

#converting area_list back to a list so I can use the in-built list
#functions
area_ls = area_ls.tolist()


#if they're going to be only two boxes in the bounding box array
#they is not point in trying to find the outliers, as one of those
#boxes will be filtered out which is not what we want as
#those boxes are most likely goig to be the boxes which will
#contain our digits
if len(bboxes) > 3:
    num_summary = self.get_five_num_summary(area_ls)
    #finding the inter quartile range of the given dataset
    IQR = self.find_IQR(num_summary)
    median = num_summary[2]

    #obtained through trial and error through all the images
    thresh_lower = 1.45
    #obtained through trial and error
    thresh_upper = 0.75

    #area can't be a negative number hence we must use the
    #abosoulte value to calculate the upper and lower bounds the
    #area can be in
    lower_bound = abs(int(median - (thresh_lower * IQR)))
    upper_bound = int(median + (thresh_upper * IQR))

    for area in area_ls:
        indx = area_ls.index(area)
        #filtering away really small boxes
        if area < lower_bound:
            #I am going to set the index of the outlier indexes to
            # -1 because if I delete the index straight away it
            #will be hard to keep a track on which index belongs to
```

```
                # which index in the list
                bboxes[indx] = [-1, -1, -1, -1]

            #filtering away from large boxes in the image
            if area > upper_bound:
                #same reasoning applied above applies here aswell
                bboxes[indx] = [-1, -1, -1, -1]

        bboxes = self.remove_invalid(bboxes)

    return  bboxes


def find_IQR(self, num_summary):
    """
    IMPORT: num_summary : a list of real numbers
    EXPORT: IQR : a real number

    PURPOSE: finding the inter-quartile range of the obtained data, so
    we can filter out the outliers in the data. The interquartile
    range is given by the differnce of the third quartile and the
    first quartile
    """

    #IQR is the difference between the lower  25 percent quartle, and
    # the upper 75 percent quartile
    IQR = num_summary[3] - num_summary[1]

    return IQR




    for indx, area in enumerate(bboxes):
        pass

def find_area(self, box):
    """
    IMPORT: box: a numpy array of datatype int32
    EXPORT: area : a real number

    PURPOSE: Is to calculate the area of a given bounding box, and we
```

```
        know regardless of the poisiton of the box the area is going
        to be always w * h
        """
        return box[2] * box[3]

    def non_max_suppression(self, bboxes):
        """
        IMPORT: bboxes : a numpy array of datatype int32
        EXPORT: bboxes : a numpy array of dataype int32

        PURPOSE: this to remove small boxes inside big boxes. This is
        specifically useful for digits such as 8 and 0, as the middle
        parts of these numbers typically get detected
        """
        #sorting boxes by the smallest area to the largest area
        bboxes = sorted(bboxes, key=lambda x: self.find_area(x))

        ##searching for boxes which are inside one another
        for curr_box_indx, curr_box in enumerate(bboxes):
            x,y,w,h = curr_box
            #ensuring that the algorithm is not comparing to itself, and
            #it's searching relative to the other boxes
            for alt_box in bboxes:
                #safe-gaurd for trying to delete an index which doesn't
                #exist in the image
                if curr_box_indx < len(bboxes):
                    #is the current box inside any of the alternate boxes
                    x_alt,y_alt,w_alt,h_alt = alt_box
                    end_point_curr_box = x + w
                    end_point_alt_box  = x_alt + w_alt

                    #if the corners of the alternate box are inside the
                    #current box then check the heights of the box
                    if x > x_alt and end_point_curr_box < end_point_alt_box:
                        #is the height of the current box inside the
                        #alternate
                        height_curr_box = y + h
                        height_alt_box = y_alt + h_alt

                        #if the height of the alternate box is inside
                        #the current box then the whol box is inside the
                        #current box hence, we can go ahead and delete this
```

```
                    #box
                    if height_curr_box < height_alt_box and \
                            y > y_alt:
                        del bboxes[curr_box_indx]

        return bboxes

    def make_new_region(self, left_box, right_box):
        """
        IMPORT:
            left_box : a numpy array of dtype int32
            right box: a numpy array of dtype int32

        EXPORT:  a numpy array of dtype int32

        PURPOSE: it's to combine two bounding boxes to make one big
        bounding box, and its main use when extracting the region of
        interest from the image

        """
        #top left corner of the left-most upper most point in the image
        #coordinates
        x = left_box[0]
        y = left_box[1]

        #right most lower most corner in the image coordiantes
        x_r =  right_box[0] + right_box[2]
        y_r = right_box[1] + right_box[3]

        #since these regions will be bounding boxes, we have to
        #calculate the width and the height of the new bounding box
        #which we have found
        w = x_r - x
        h = y_r - y

        return np.array([x, y, w, h], dtype='int32')



    def show_debug_boxes(self, bboxes, im, title):
        """
        IMPORT:
```

```
        im :  a numpy array of dtype int32
        title : string
        EXPORT: none

        PURPOSE: the module is meant to help with debugging purposes, and
        it's meant to show what's happening at each step of the image
        segmentation algorithm
        """
        debug_im = im.copy()
        blue = (255,0,0)
        self.draw_boxes(bboxes, debug_im, blue)
        cv.imshow(title, debug_im)
        cv.waitKey()
        cv.destroyAllWindows()

    #FUNCTIONS WHICH WILL HELP TO FIND THE INTEREST AREA
    def resize_image(self, im, x, y):
        """
        IMPORT:
            im :  a numpy array of dtype int32
            x : an integer number
            y : an integer number

        EXPORT: im:  a numpy array of dtype int32

        PURPOSE: it's to resize the image to the given specification of
        x and y of the image
        """
        return cv.resize(im, (int(x), int(y)))


    def group_clusters(self, clusters):
        """
        IMPORT: clusters:  a numpy array of dtype int32
        EXPORT: clutesrs:  a numpy array of dtype int32

        PURPOSE: many boxes where found by the find_clusters algorithm,
        this function responsibility it to clean up the bounding boxes
        found hence, to select the biggest box out of the cluster, and
        to make this the new box for that section of the image
        """
        cluster_b = []
```

```
        #the bboxes are put into pairs whereby each pair is very close to
        #each other hence, we can just sort by first box as that box will
        #be left most box out of the two boxes
        clusters = sorted(clusters, key=lambda x: x[0][0])

        for indx, cluster in enumerate(clusters):
            box_one = cluster[0]
            box_two = cluster[1]

            x_1, y_1, w_1, h_1 = box_one
            x_2, y_2, w_2, h_2 = box_two

            #getting the longer lines out of the two boxes parsed in
            nw_h = max(h_1, h_2)
            nw_w = max(w_1, w_2)

            #getting the left-most x and y points so we know where the
            #clusters are going to begin
            nw_x = min(x_1, x_2)
            nw_y = min(y_1, y_2)


            #the new box is going to be the bigger box out of the cluster
            #which we just found
            nw_box = np.array([nw_x, nw_y, nw_w, nw_h], dtype='int32')
            clusters[indx] = nw_box
        return clusters

    def  find_clusters(self, bboxes, thresh_x, thresh_y):
        """
        IMPORT:
            bboxes:  a numpy array of dtype int32
            thresh_x : integer
            thresh_y : integer

        EXPORT: cluster :  a numpy array of dtype int32

        PURPOSE: the numbers which are in the image should be relatively
        close to each other hence, we're going to get rid of all the boxes
        which are not close to each other because the chance of these
        boxes not been a number is very high
        """
```

```
cluster = []

#sorting the bounding boxes from the leftmost box to the right
#most box
bboxes = sorted(bboxes, key=lambda x: x[0])
bboxes = self.remove_invalid(bboxes)

for start, curr_box in enumerate(bboxes):
    x,y,w,h = curr_box
    pt1 = (x, y)
    pt2 = (x+w, y+h)
    for alt_box in bboxes:
        x_alt,y_alt,w_alt,h_alt = alt_box
        pt1_alt = (x_alt,y_alt)
        pt2_alt =(x_alt+w_alt, y_alt+h_alt)

        #seeing what the gap is between the current box
        #and the alternate box. Hence, we  grapbbed the images
        #from left to right the gap is alway going to be
        #calculated this way
        x_diff = abs(pt2[0] - pt1_alt[0])
        #finding the gap between bounding boxes in the vertical
        #direction
        y_diff = abs(pt2[1] - pt2_alt[1])

        #getting the longest width and height, so we can use
        #those values to calculate our tolerance. We want to do
        #this because out tolerance will change with the size of
        #image instead of having a hard coded value
        line_seg_x = max(w, w_alt)
        line_seg_y = max(h, h_alt)

        line_TOL_x  = line_seg_x * thresh_x
        line_TOL_y = line_seg_y * thresh_y

        #if the gap in the horizontal, and vertical direction is
        #less than the tolerance this is most likely going to be a
        #cluser of boxes
        if x_diff <= line_TOL_x:
                if y_diff <= line_TOL_y:
                    pair = [curr_box, alt_box]
                    pair = sorted(pair, key=lambda x: x[0])
```

```
                        cluster.append([curr_box, alt_box])
    return cluster

def filter_bounding_boxes(self, bboxes, lower_thresh=1.10, upper_thresh=3
    """
    IMPORT:
        bboxes:  a numpy array of dtype int32
        lower_thresh : real number
        upper_thresh : real number

    EXPORT: bboxes :  a numpy array of dtype int32

    PURPOSE: we know that for the bounding boxes which will contain
    the digits the height is going to be longer than the width relative
    to a ratio. Hence, for bounding boxes which exceed this ratio
    they should be filtered out and discarded
    """
    #sorting the bounding boxes from the leftmost to the right most of
    #of the image
    bboxes = sorted(bboxes, key=lambda x: x[0])

    for indx, box in  enumerate(bboxes):
        x,y,w,h = box
        pt1 = (x, y)
        pt2 = (x+w, y+h)

        ratio = h/w
        #we're going to expect the height of the digits to be no more
        #than than the width of the bounding box hence filter
        #boxes which will violate that expectation
        if ratio < lower_thresh or ratio > upper_thresh:
            bboxes[indx]  = [-1, -1, -1, -1]

    bboxes = self.remove_invalid(bboxes)

    return bboxes

def find_leftmost_pt(self, bboxes, reverse=False):
    """
    IMPORT:
        bboxes :  a numpy array of dtype int32
        reverse : boolean
```

```
    EXPORT: points of a bounding box which will contain the
    left most points

    PURPOSE: to find the left upper most point of a given set of
    boudngind boxes if reverse is true, it will find the right-most
    lower most point of the bounding boxes

    Limitations of this algorithm
        - if a box is inside another box, it's going to find the box
        inside because it's comparing in relation to the left point of
        the box
    """
    bboxes = self.remove_invalid(bboxes)
    #sorting the bounding boxes from left most to the right most of the
    #image
    left_most_boxes = sorted(bboxes, key=lambda x: x[0], reverse=reverse)

    #the likely left most box although we have to check if they're
    #going to be boxes above this box
    temp_box = left_most_boxes[0]

    #CASE 1: clear left most box will be met if it fails CASE 2's and
    #CASE 3's checks

    #CASE 2: boxes are the same x-coordinate hence to enusre that
    #the upper-most box is selected
    for box in left_most_boxes:
        #case: when two boxes have the same x-dimension but differing
        #y-dimensions
        if temp_box[0] == box[0]:
            if temp_box[1] > box[1]:
                temp_box = box

    #CASE 3: the left most box is selected but if they's a box which is
    #higher than the current box combine find the intersecting points
    highest_boxes = sorted(bboxes, key=lambda y: y[1], reverse=reverse)
    highest_box = highest_boxes[0]

    equal = highest_box == temp_box
    #if the current box is not the highest box, form an intersection
    #with the highest box
```

```
        if not equal.all():
            temp_box = self.find_intersection(highest_box, temp_box,
                    reverse=reverse)

        if self._DEBUG:
            print('='*80)
            print(red, 'find_leftmost_p() | temp box',reset)
            print('\t {}'.format(temp_box))
            print('='*80)

        return temp_box[0], temp_box[1], temp_box[2], temp_box[3]

    def remove_invalid(self, bboxes):
        """
        IMPORT: bboxes:  a numpy array of dtype int32
        EXPORT: bboxes :  a numpy array of dtype int32

        PURPOSE: to move any bounding box which has set to all -1 from
        any of the filtering algorithms found in this file
        """
        if self._DEBUG:
            print('='*80)
            print(red + "og array" + reset, bboxes)
            print('='*80)


        nw_bboxes = []
        for indx, box in enumerate(bboxes):
            #if the first index is equal to -1 the whole box will equal to
            #-1,and that will be an invalid box
            if box[0] == -1:
                #ignoring every single box which has a negative one
                #as its first index
                pass
            else:
                nw_bboxes.append(box)


        #converting the new list into an array, so openCV function can use
        #this array to draw the boxes
        bboxes = np.array(nw_bboxes, dtype='int32')
```

```
        if self._DEBUG:
            print('='*80)
            print(red + "resultant array" +reset, bboxes)
            print('='*80)

        return bboxes

    def find_intersection(self, box_one, box_two, reverse=False):
        """
        IMPORT:
            box_one :  a numpy array of dtype int32
            box_two :  a numpy array of dtype int32
            reverse: boolean

        EXPORT: a bounding box which is a numpy array

        PURPOSE: to find the intersection between two boxes, this will
        by defualt find the intersection on the left side of the bounding
        boxes. Hence, to find the intersetion on right side of the bounding
        boxes set reverse to true
        """
        temp_boxes = [box_one, box_two]
        #placing the box with the lowest x value at the front
        temp_boxes = sorted(temp_boxes, key=lambda x: x[0], reverse=reverse)
        #the first boxes x coordinate
        nw_x = temp_boxes[0][0]
        #the right most point will be the temp box's in reverse, and it
        #will be the that boxes x value plus that boxes w value
        nw_w = temp_boxes[0][2]
        #placing the box withthe lowest y value at the front
        temp_boxes = sorted(temp_boxes, key=lambda  y: y[1], reverse=reverse)
        #the first boxes y coordinate
        nw_y = temp_boxes[0][1]
        #the right most point will be the temp boxes in reverse, and it
        #will be that  box's y value plus box's h value
        nw_h = temp_boxes[0][3]

        if self._DEBUG:
            print('='* 80)
            print(red, 'find_intersection() | interesction',reset)
            print('\t {}, {}, {}, {}'.format(nw_x, nw_y, nw_w, nw_h))
            print('='* 80)
```

```
        return nw_x, nw_y, nw_w, nw_h

    def draw_boxes(self, bboxes, im, color=(0,0,255)):
        """
        IMPORT:
            bboxes:  a numpy array of dtype int32
            im:  a numpy array of dtype int32
            color : a numpy array of three integers which will represent
            the color

        EXPORT: none

        PURPOSE: to draw a list of bounding boxes onto the image. This
        function is mainly for vidualisation purposes so the user can
        see what's going on at each stage of the algorithm in
        realtion to the produced bounding boxes
        """
        for box in bboxes:
            #if the box has been labelled by a negative -1 by a filtering
            #algorithm we should skip over this box
            if box[0] == -1:
                pass
            else:
                x,y,w,h = box
                cv.rectangle(im, (x,y), (x+w, y+h), color, 2)

    def _validate_image(self, in_im):
        """
        IMPORT: in_im : a numpy array of dtype int32
        EXPORT: in_im :  a numpy array of dtype int32

        PURPOSE: to determine if the thing which is  been passed into
        the image class is actually an image, and if it's the
        right data types for images
        """
        #an image is going to be an numpy matrice
        if not type(in_im) == np.ndarray:
            #all loaded images, are an unsigned interger by defualt
            if not in_im.dtype == 'uint8':
                raise ImageError("Error: an image wasn't laoded in the system
```

```
    return in_im
```

D. ImageLoader.py

```
"""
AUTHOR: Tawana Kwaramba: 19476700
LAST EDITED:

PURPOSE OF FILE: A class which will allow us to load the images for efficientl
and effectively. This class will allow you to store all the paths for a given
path and will only load the images when needed instead of loading all the ima
in memory at once.
"""

import os
import cv2 as cv
from Errors import *

class Image_Loader(object):
    """
    CLASS WAS ADAPTED FROM: Kuklin, Maxim. 2020. "Efficient image loading".
    Learn OpenCV. https://www.learnopencv.com/efficient-image-loading/
    """

    ext = (".png", ".jpg", ".jpeg")
    modes = ('RGB', 'HSV', 'LUV', 'BGR', 'LAB', 'GRAY')

    #choosing HSV as the defualt color channel as this typically gives better
    #results for image segementation
    def __init__(self, path, mode="HSV"):
        self._path = path
        self._mode = mode
        self._data = self.load(self.path)
        self._image_indx = 0

    #=========================ACCESORS=====================================
    @property
    def path(self):
        return self._path

    @property
    def mode(self):
        return self._mode
```

```
    @property
    def data(self):
        return self._data

    @path.setter
    def path(self, nw_path):
        self._path = nw_path
        #self._path = self.__validate_path(nw_path)
        #re-loading the data at the new path
        self._data = self.load(self.path)

    #============================SETTERS=====================================
    @mode.setter
    def mode(self, nw_mode):
        self._mode = self.__validate_mode(nw_mode)

    #============================PUBLIC METHODS==============================
    def load(self, path):
        """
        IMPORT: path (string)
        EXPORT: res (list)

        ASSERT: returns a list of image file[s]
        """
        if os.path.isfile(path):
            #res has to be the same data type as the return of load_dir so I
            #can access the images the same way regardless if it's a file or
            #a directory
            res = [path]

        if  os.path.isdir(path):
            res = self.load_dir(path)

        return res

    def load_dir(self, path):
        """
        IMPORT: path (string)
        EXPORT: paths (list)

        PURPOSE: returns a list of relitive paths of image files inside a
        specified directory
```

```
        """
        #getting all the images insde the folder as a list and returning it
        all_files = os.listdir(path)
        paths = [os.path.join(path, image) for image in all_files]

        return paths


    def load_image(self, path):
        """
        IMPORT: path (string)
        EXPORT: convert_img (numpy matrix)

        PURPPOSE: to load the image found in the specified path given the
        specified mode to load the image in
        """
        convert_img = cv.imread(path)
        if self._mode == 'GRAY':
            convert_img = cv.cvtColor(convert_img,cv.COLOR_BGR2GRAY)
        elif self._mode == 'HSV':
            convert_img = cv.cvtColor(convert_img,cv.COLOR_BGR2HSV)
        elif self._mode == 'LUV':
            convert_img = cv.cvtColor(convert_img,cv.COLOR_BGR2Luv)
        elif self._mode == 'LAB':
            convert_img = cv.cvtColor(convert_img,cv.COLOR_BGR2Lab)
        elif self._mode == "RGB":
            convert_img = cv.cvtColor(convert_img,cv.COLOR_BGR2RGB)

        return convert_img

    def create_labels(self):
        """
        EXPORT: labels (list)
        PURPOSE: to create labels from the loaded directories. Hence, it
        will make the labels the same as the directory names
        """
        labels = []
        for path in self._data:
            consititutes = path.split('/')
            #the name of the directory is always going to be the last index
            #of the splitted list
            labels.append(consititutes[-1])
```

```
        return labels


    def __iter__(self):
        """
        EXPORT: reference to this object

        PURPOSE: Defining how this class should be initialised when another f
        tries to iterate over this class
        """
        self._image_indx = 0
        return self

    def __len__(self):
        """
        EXPORT: the lenth of the data  class field
        PURPOSE: Defining how this class should behave when the user calls th
        len() function on this class
        """
        return len(self._data)

    def __next__(self):
        """
        EXPORT: the image matrix at the given index

        PURPOSE: Defining what this class should return and behave when the c
        is used in a for-each loop
        """
        if self._image_indx == len(self._data):
            raise StopIteration
        curr_img_path = self._data[self._image_indx]
        im = self.load_image(curr_img_path)

        self._image_indx += 1


        return im

    def __validate_path(self, nw_path):
        """
        IMPORT: nw_path (string)
```

```
        EXPORT: nw_path (string)

        PURPOSE: to validate if the passed nw_path string is not an empty
        string and it's a string, to ensure that the path is an actual image
        file or it's a directory
        """
        if (nw_path is None or nw_path == "")  and not(isinstance(nw_path,str
            raise PathError("""
            path path can't be an empty string or not a string:
            input:
            {}
            type:
            {}""".format(nw_path, type(nw_path)))

        if not (os.path.isfile(nw_path) or os.path.isdir(nw_path)):
            raise PathError("path is not a valid file or directory: %s"
                    %nw_path)

        if (os.path.isfile(nw_path)):
            if not nw_path.lower.endswith(self.ext):
                raise PathError("not recognised file extension %s" % nw_path)

        return nw_path

    def __validate_mode(self, nw_mode):
        """
        IMPORT: nw_mode (string)
        EXPORT: nw_mode (string)
        PURPOSE: to validate if the new mode is a valid mode specified by the
        class
        """
        if nw_mode.upper().strip() not in self.modes:
            raise modeError("""
            mode type of:
            %s
            the mode must be one of these types:
            %s
            """ % (nw_mode, self.modes))
        return nw_mode
```

E. Colours.py

```
"""
AUTHOR: Tawana Kwaramba: 19476700
LAST EDITED:

PURPOSE OF FILE: is to contain all the ANSI escape codes correponding to
colour change int the terminal.

The main rationale for this file is to be able to make the outputted text
in the terminal for readable. So the errors, warnings, and success will
stand out more when running the programme
"""
green = '\033[32m'
reset = '\033[m'
red = '\033[31m'
yellow = '\033[33m'
```

F. Errors.py

```
"""
AUTHOR: Tawana Kwaramba: 19476700
LAST EDITED:

PURPOSE OF FILE: it's to define custom exception. So it allows the user to be
able to quickly diagnose the error in the code
"""
from Colours import *

class Error(Exception):
    pass

class PathError(Error):
    """
    Error raised when the path is not a string or the path is empty.
    Furthermore, this error can be raised if the path is not a file or a
    directory
    """
    def __init__(self, mssg):
        self.mssg = red + "ERROR " + reset + mssg


class modeError(Error):
    """
    ERROR raised when the user tries to load  in a mode which is not
    recognised by the programme
    """
    def __init__(self, mssg):
        self.mssg = red + "ERROR " + reset + mssg


class ImageError(Error):
    """
    ERROR raised when the user tries to load  in a mode which is not
    recognised by the programme
    """
    def __init__(self, mssg):
        self.mssg = red + "ERROR " + reset + mssg
```