INDIAN INSTITUTE OF ENGINEERING SCIENCE AND TECHNOLOGY, SHIBPUR
Howrah, West Bengal, India - 711103

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

A MINI-PROJECT REPORT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

ON

"OBJECT RECOGNITION TO KEEP SOCIAL DISTANCING NORMS IN CHECK"

SUBMITTED BY

Abhinaba Chowdhury (510519007)
Abhiroop Mukherjee (510510109)
Debarghya Dey (510519087)
Jyotiprakash Roy (510519016)
Shrutanten (510519048)

UNDER THE GUIDANCE OF

DR. SAMIT BISWAS

(Academic Year: 2020-2021)

INDIAN INSTITUTE OF ENGINEERING SCIENCE AND TECHNOLOGY, SHIBPUR
Howrah, West Bengal, India - 711103

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

Certificate

It is certified hereby that this report, titled *"Object Recognition to keep Social Distancing Norms in check"*, and all the attached documents herewith are authentic records of Abhinaba Chowdhury (510519007),
Abhiroop Mukherjee (510510109), Debarghya Dey (510519087), Jyotiprakash Roy (510519016), and Shrutanten (510519048) from the Prestigious Department of Computer Science And Technology of the Distinguished and Respected IIEST Shibpur under my guidance.

The works of these students are satisfies all the requirements for which it is submitted. To the extent of my knowledge, it has not been submitted to any different institutions for the awards of degree/diploma.

Dr. Samit Biswas
Asst. Professor

Dr. Sekhar Mandal
Head Of Department

ACKNOWLEDGEMENT

We, as the students of IIEST, consider ourselves honoured to be working with Dr. Samit Biswas. The success of this project would not have been possible without his useful insights, appropriate guidance and necessary criticism.

We would pass our token of token of gratitude to the Department of Computer Science And Technoogy as well for providing us with the opportunity to be able to tackle real world problems while improving our problem solving ability and thinking capacity by organising this project. We all have learnt quite a handful of new skills and are eager to use them henceforth as well.

Abhinaba Chowdhury (510519007)
Abhiroop Mukherjee (510510109)
Debarghya Dey (510519087)
Jyotiprakash Roy (510519016)
Shrutanten (510519048)

Contents

List of Figures

1 INTRODUCTION

1.1 Motivation

- Coronaviruses are a group of related RNA viruses that cause diseases in mammals and birds. In humans and birds, they cause respiratory tract infections that can range from mild to lethal. Mild illnesses in humans include some cases of the common cold (which is also caused by other viruses, predominantly rhinoviruses), while more lethal varieties can cause SARS, MERS, and COVID-19.
- With the increase in the spread of the dangerous and highly contagious **Novel Coronavirus** and the underlying disease caused by it, **COVID-19**, it is a requirement now more than ever to follow the social distancing norms set in place by the scientists and researchers.
- But as we all know, India is a country with a not-so-small population, so it is pretty understandable and obvious that the law enforcement will not be able to actually enforce it on every single person. Therefore, new means of automata in place of actual individuals is a no brainier.
- **That is where we come in.**

1.2 The Idea Behind The Project

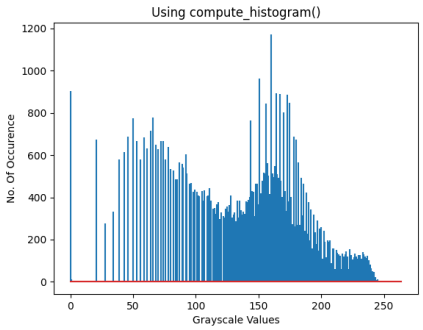
The idea behind the working of this software was simple. The software just needed to be able to look at a live feed (or recorded footage) of a camera and know which of the people present in the footage are actually following the social distancing norms and which of them are not, and mark either one appropriately. That is where our journey to build a social distance checker started.

2 KNOWLEDGE REFINEMENT

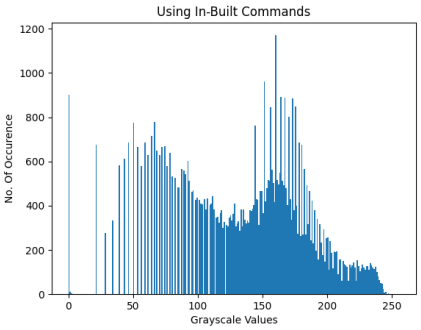
Before we settled on the topic of object detection and started building this project, we got some practice, which was necessary since we were going to dip our toes in image processing.

- We made histograms for grey-level images. This was done using both the OpenCV's `ravel()` function and our own implementation of it, called `compute_histogram()`.

(a) Greyscale Image



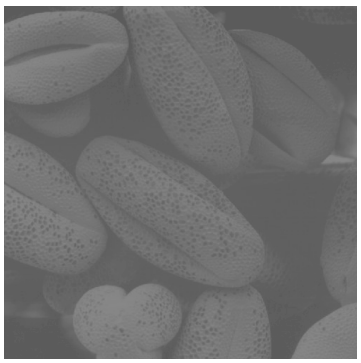
(b) histogram by `compute_histogram()`



(c) histogram by in-built `ravel()`

Figure 1: Histogram of Greyscale Images

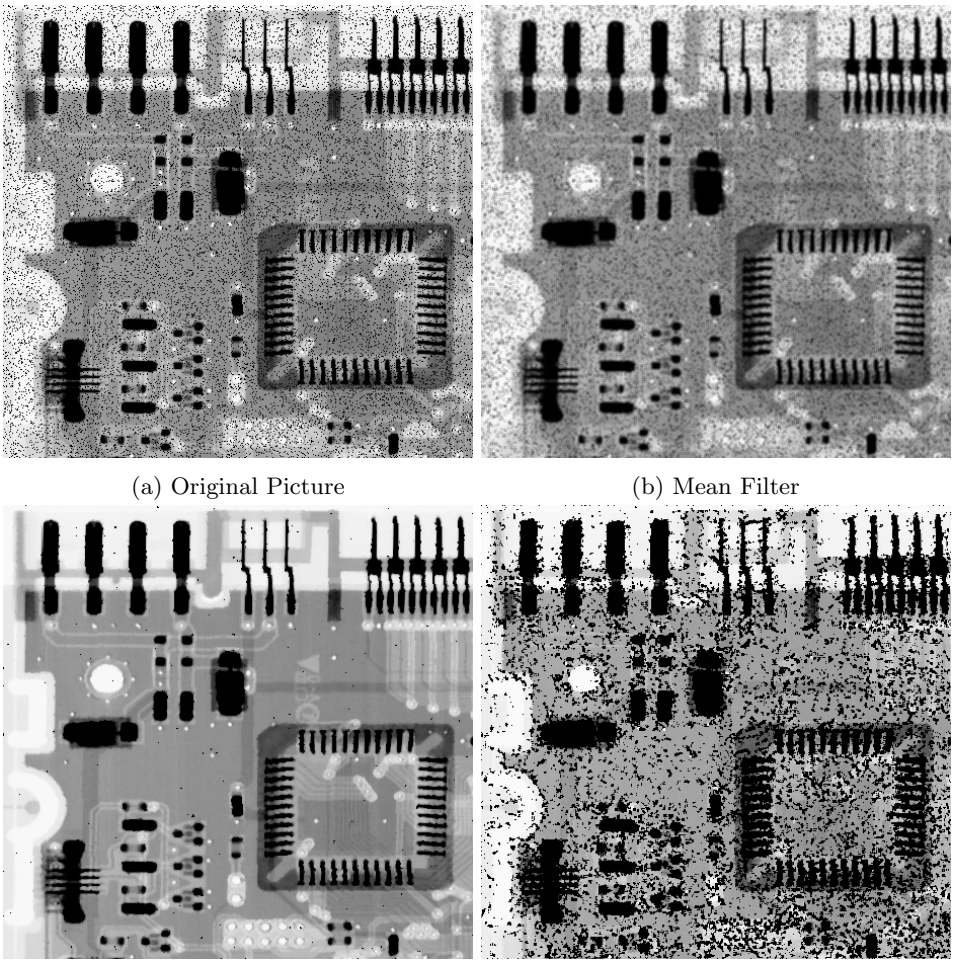
- Once that was over, we moved onto some Image Enhancement skills. Here we implemented noise reduction functions using mean, mode and median filters.



(a) Original Image

(b) Edited Image

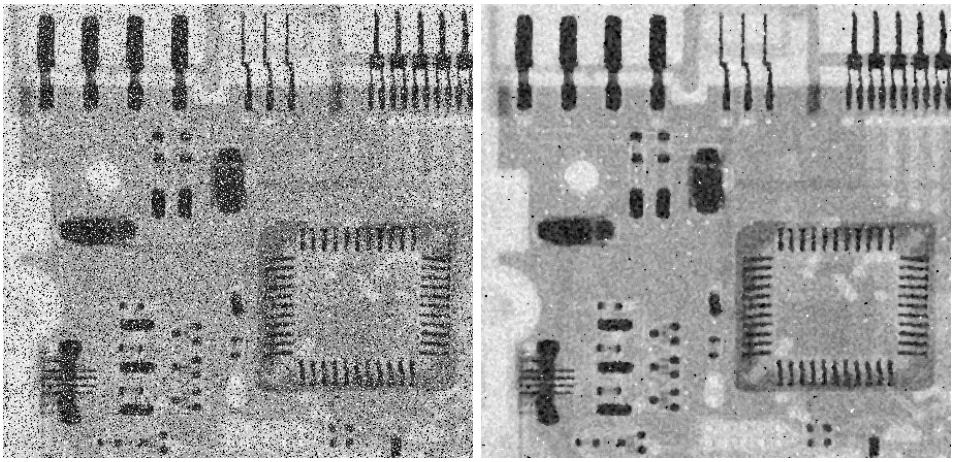
Figure 2: Contrast Enhancement using Histogram Equalization



(a) Original Picture (b) Mean Filter

(c) Median Filter (d) Mode Filter

Figure 3: Mean, Median, and Mode Filter



(a) Original Photo (b) Edited Photo

Figure 4: Salt And Pepper Noise Removal

- Then we implemented Otsu's Thresholding Algorithm [1] using minimization of *within class variance approach.*

```
import matplotlib.pyplot as plt
import cv2 as cv
import numpy as np

def otsu_threshold (img):
    hist = cv.calcHist([img],[0],None,[256],[0,256])
    hist_norm = np.divide(hist.ravel(),hist.sum())
    q = hist_norm.cumsum()

    bins = np.arange(256)

    fn_min = np.inf
    threshold = -1

    for i in range(1,256):
        p1,p2 = np.hsplit(hist_norm,[i])

        q1 , q2 = q[i] , q[255]-q[i]

        if q1 < 1.e-6 or q2 < 1.e-6:
            continue

        b1 , b2 = np.hsplit(bins,[i])

        m1 = np.sum(p1 * b1)/q1
        m2 = np.sum(p2 * b2)/q2

        v1 = np.sum( ((b1 - m1)**2) * p1)/q1
        v2 = np.sum( ((b2 - m2)**2) * p2)/q2

        fn = v1*q1 + v2*q2

        if fn < fn_min:
            fn_min = fn
            threshold = i

    return threshold



img = cv.imread("input.jpg",0)

threshold = otsu_threshold(img)

a, img_my = cv.threshold(img,threshold,255,cv.THRESH_BINARY)
cv.imshow("My",img_my)

ret , img_os = cv.threshold(img,0,255,cv.THRESH_BINARY + cv.THRESH_OTSU)
cv.imshow("OS",img_os)

#cv.imwrite("output.jpg",img_my)

cv.waitKey(0)
```

Listing 1: Our Implementation Of Otsu's Thresholding Algorithm

3 PREREQUISITES

3.1 Outdoor Requirements

It is important to mention here that this is not a portable software that can be fed any footage and just be expected to work. There need to be some calibration measures taken to actually get this software working:

- Actually knowing the local social distancing norms
  - The minimum distance set for social distancing by the local government
- Finding a good position for the camera
  - The footage needs to be taken from a high enough place
- Knowing the required distance in pixels
  - This will depend on the position and angle of the camera's view

3.2 Hardware and Software Requirements

The tools used to build this software are platform independent. However, there are a few requirements needed to be fulfilled to get the program working. These are:

- Software Requirements
  - Python - 3.5 or above
  - OpenCV-Python - version 2 or above
  - YOLOv3 Configuration and Network Weights
  - Numpy
- Hardware Requirements
  - A CUDA enabled GPU is optional yet recommended to get the best performance.
  - If such a GPU is not being used, the CPU needs to be good enough.

4 THE PROJECT

4.1 Software Used

The softwares used to build this *checker* are:

4.1.1 An Integrated Development Environment (IDE)

An Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of at least a source code editor, build automation tools and a debugger. Some IDEs contain the necessary compiler, interpreter, or both; others, do not.

We have used PyCharm as our IDE, as it was easy to set up and write code in.

PyCharm is an integrated development environment (IDE) used in computer programming, specifically for the Python language. It is developed by the Czech company JetBrains. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems (VCSes), and supports web development with Django as well as data science with Anaconda.

4.1.2 Python

Python is an interpreted, high-level and general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Why did we choose Python?

1. Python has an upper hand when it comes to software based on image recognition and object detection. Since it is the main objective of the project, choosing python was a given.Python has an upper hand when it comes to software based on image recognition and object detection. Since it is the main objective of the project, choosing python was a given.
2. Python is unbeaten when it comes to Machine Learning. Python has support for myriad machine learning libraries, such as OpenCV, the one being used here.
3. Python is comparatively easier to understand and learn. The syntax is clear and simple to read and write.
4. And just our overall experience of using python for years.

4.1.3 Google Colab

After working on the project for quite some time, we realised that we did not have enough hardware resources at out disposal to actually make the *checker* work smoothly. So we decided on shifting to Google Colab. Google Colab is an online iPython development environment similar to Jupyter Notebook. It uses CUDA acceleration to speed up processes, so we switched to it rather than continuing development locally.

4.1.4 LaTeX

LaTeX was used to write this report. LaTeX is a software system for document preparation. When writing, the writer uses plain text as opposed to the formatted text found in "What You See Is What You Get" word processors like Microsoft Word or LibreOffice Writer.

4.2 The Program

4.2.1 Outline

The blueprint of this *checker* that we thought of initially:

1. Video Input
   - Need some way to handle video input coming through the camera feed
2. Processing
   - The input needs to be processed somehow
3. Detecting people
   - Need to identify people in the video feed
4. Measuring distance between each couple
   - Need to calculate the distance between every two persons
5. Mark the violations
   - Need to mark the ones that violate social distancing norms

4.2.2 Proceedings

How we proceeded with the outlines of the blueprint:

1. Video Input

   This was easier than we expected it to be. We just had to get our hands on some recorded footage of somewhat populated areas. We refrained from using live footage because:
   - It is tough to get our hands on the light footage of a security camera or the equivalent.
   - If the checker worked on recorded footage, it would work on live footage as well.

   The videos we ended up choosing:

(a) Pedestrian Video (b) Shibuya Video

Figure 5: Still Pictures from Sample Videos

2. Processing

We used the OpenCV library for our video/image processing. It is a really handy library that can be used for image processing, object detection and many other purposes.

```python
import cv2 as cv

def total_frames(file_name):
    cap = cv.VideoCapture(file_name)
    res = 0

    while True:
        ret,img = cap.read()

        if not ret:
            break

        res = res+1

    return res

file_name = "pedestrian.mp4"
tot_frame = total_frames(file_name)
print(f"Total Frames in {file_name} are {tot_frame}")
```

Listing 2: A sample code to count no. of frames in a video

3. Detecting People

- For this we decided to go with the You Only Look Once (YOLO) algorithm for object detection. The algorithm itself is discussed a bit later in the report.
- We did not train the object detection neural network model ourselves. We used the prebuilt model, trained by the Darknet team because of time constraints.
- The script that we wrote for our implementation of the YOLO algorithm:

```python
import cv2 as cv #OpenCV Library
import numpy as np  #for handling arrays

# Give the configuration and weight files for the model and load the network.
net = cv.dnn.readNetFromDarknet('yolov3.cfg', 'yolov3.weights')  # Reads Network from .cfg and .weights
net.setPreferableBackend(cv.dnn.DNN_BACKEND_CUDA)   # this specifies what type of hardware to use (GPU or CPU)
net.setPreferableTarget(cv.dnn.DNN_TARGET_CUDA)     # sets preferable hardware

threshold = 0.75
nms_threshold = 0.012
distance_px=120   # arbitrary value for now but looked best

def object_detection_YOLO(img,threshold,nms_threshold):
    # determine the output layers
    ln = net.getLayerNames()
    ln = [ln[i[0] - 1] for i in net.getUnconnectedOutLayers()]
    # construct a blob from the image
    # blob is just a preprocessed image
    blob = cv.dnn.blobFromImage(img, 1/255.0, (416, 416), swapRB=True, crop=False)      # blob = boxes
    # blobs goes as the input to YOLO
    # inputting blob to the Neural Network
    net.setInput(blob)
    # t0 = time.time()
    outputs = net.forward(ln)   # finds output
    # t = time.time()
    # print('time=', t-t0)

    boxes = []
    confidences = []
    centroids = []
    results = []

    h, w = img.shape[:2]
```

```
    for output in outputs:  # Outputs have all the detection and their probability for every class
        for detection in output:    # detection is the the list of all probabilities with box dimension in start
            scores = detection[5:]  # everything in array after 5th element
            classID = np.argmax(scores)     # picks the maximum probability
            confidence = scores[classID]

            if (confidence > threshold) and (classID == 0):
                #first 4 elements are box characteristics normalized to range(0,1)
                #first two element are middle co-ordinate
                # next two are width and height of blob
                box = detection[:4] * np.array([w, h, w, h])
                (centerX, centerY, width, height) = box.astype("int")   # typecasting to int, as array indexes are int
                x = int(centerX - (width / 2))      # finding upper corner
                y = int(centerY - (height / 2))
                box = [x, y, int(width), int(height)]   # changing origin to top left and typecasted to int
                boxes.append(box)                       # added the box to boxes
                confidences.append(float(confidence))   # added confidence to confidences
                centroids.append((centerX,centerY))

    indices = cv.dnn.NMSBoxes(boxes, confidences,score_threshold=threshold,nms_threshold=nms_threshold)
    # score_threshold -> threshold for confidence
    # nms_threshold -> threshold for how close to blobs are, if two blobs are too close, one of them is discarded
    # closeness is determined by IoU (intersection over Union)
    # discarding is based on confidence, higher confidence is retained

    boxes_final=[]; confidences_final=[]; centroids_final=[]
    if len(indices):
        for i in indices.flatten():
            # extract the bounding box coordinates
            x, y = (boxes[i][0], boxes[i][1])
            w, h = (boxes[i][2], boxes[i][3])
            boxes_final.append((x,y,w,h))
            confidences_final.append(confidences[i])
            centroids_final.append(centroids[i])

    return boxes_final,confidences_final,centroids_final
```

Listing 3: Our implementation of object (people) detection using YOLO

4. Measuring distance between each couple

   - This was undeniably the toughest part of the project and took the longest time. First we decided to go with measuring the Euclidean distance between the centroids of every two detections. But that may not work in every condition since it depends on the placement of camera and the viewing angle from the ground and the angle from the perpendicular to the ground.
   - A conversion of the 3-dimensional footage being fed to the algorithm to 2-dimensions was more than necessary to get the top view of every frame to avoid the *viewing angle problem*. The *viewing angle problem* can be defined as the enigma that arises while trying to measure distances without knowing the angle between the object's line of sight and the ground.

- Enter **Bird's Eye View (BEV)**. This is what we call the top view of every frame. This was made possible by OpenCV's `getPerspectiveTransform()` and `warpPerspective()` functions.

```
def birds_eye_view(corner_points,width,height,image):
    """
    Compute the transformation matrix
    corner_points : 4 corner points selected from the image
    height, width : size of the image
    return : transformation matrix and the transformed image
    """
    # Create an array out of the 4 corner points
    corner_points = np.float32(corner_points)
    # Create an array with the parameters (the dimensions) required to build the matrix
    img_params = np.float32([[0,0],[width,0],[0,height],[width,height]])
    # Compute and return the transformation matrix
    matrix = cv.getPerspectiveTransform(corner_points,img_params)
    img_transformed = cv.warpPerspective(image,matrix,(width,height))

    return matrix,img_transformed
```

Listing 4: Function Bird's Eye Perspective Transformation Matrix

- This piece of code essentially calculates what is called a *transformation matrix*[2] for the supplied image (frame) which can then be used to get the centroids of the points as seen from a vertical position directly above the center of the rectangle passed to the function.

```
def birds_eye_point(matrix,centroids):
    """ Apply the perspective transformation to every ground point which have been detected
     on the main frame.
    @ matrix : the 3x3 matrix
    @ centroids : list that contains the points to transform
    return : list containing all the new points
    """
    # Compute the new coordinates of our points
    points = np.float32(centroids).reshape(-1, 1, 2)
    transformed_points = cv.perspectiveTransform(points, matrix)
    # Loop over the points and add them to the list that will be returned
    transformed_points_list = list()

    for i in range(0,transformed_points.shape[0]):
        transformed_points_list.append([transformed_points[i][0][0],transformed_points[i
    ][0][1]])

    return transformed_points_list
```

Listing 5: Function which converts coordinates to its bird's view coordinates

- We used these functions to get a two dimensional view of every frame and calculate distance between every pair of detections (people).

5. Mark the violations

   This was again a fairly easy step. We just needed the coordinates of the people in the *violation zone* and make their detection rectangle red as opposed to green.

6. The final result

   We were able to process video in around **10 fps**. The end result of days of hard work and patience was the following program:

```
import cv2 as cv #OpenCV Library
import numpy as np  #for handling arrays
#from tqdm.std import tqdm  #for system progressbar
from tqdm.notebook import tqdm #for google colab progressbar
from scipy.spatial import distance  #for cdist
from time import time # for testing purposes

############################ File Setup ##############################
file_name = "pedestrian.mp4"
tot_frame = total_frames(file_name)
```

```
cap = cv.VideoCapture(file_name)

fourcc = cv.VideoWriter_fourcc(*'mp4v')
out = cv.VideoWriter('output.mp4',fourcc, 25.0,(1282,400))

############################ Bird Eye Transform Setup ##############################
corners=[(775,10),(1270,60),(0,350),(1100,700)]  # these are the best looking coordinate I got via hit and trial

tl,tr,bl,br=corners   #top-left, top-right, bottom-left, bottom-right

width1 = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
width2 = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
width_final = max(int(width1), int(width2))

height1 = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
height2 = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
height_final = max(int(height1), int(height2))

static_frame=cv.imread("static_frame_from_video.jpg")
static_frame=cv.resize(static_frame,(1280,720))  # these are the dimensions we are using for the video

persp_matrix, transformed_img = birds_eye_view(corners, width_final, height_final, static_frame)

############################ Making Heading ##############################

camera_view_heading = np.zeros((40,640,3),np.uint8)
camera_view_heading_text = "Camera View"
white = (255,255,255)
camera_view_heading = cv.putText(camera_view_heading, camera_view_heading_text, (220, camera_view_heading.shape[0]-13), cv.FONT_HERSHEY_SIMPLEX,0.85, white,2)

bird_eye_heading = np.zeros((40,640,3),np.uint8)
bird_eye_heading_text = "Bird-Eye View"
bird_eye_heading = cv.putText(bird_eye_heading, bird_eye_heading_text, (220, bird_eye_heading.shape[0]-13), cv.FONT_HERSHEY_SIMPLEX,0.85, white,2)

############################ Process The Input ##############################
initial_time=time()
for i in tqdm (range (tot_frame), desc="Processing..."):
    ret,img = cap.read()
    if not ret: break

    birds_display=cv.warpPerspective(img,persp_matrix,(width_final,height_final))

    boxes,confidences,centroids=object_detection_YOLO(img, threshold, nms_threshold)
    # box -> x,y,w,h
    # confidence -> confidence of the detected object
    # centroid -> center of the bbox, 2 values list

    detections=len(boxes)

    # violate=set()  # instead of a set, we can use a dictionary to speed stuff up.
    violate={}

    if detections>1:  # to check if there are at least two people in the frame, otherwise no need to run the algorithm

        transformed_centroids=birds_eye_point(persp_matrix,centroids)
        transformed_centroids=np.array([(int(x),int(y)) for x,y in transformed_centroids])

        # calculates the distance between all the pairs of points
        D=distance.cdist(transformed_centroids,transformed_centroids,metric="euclidean")

        for i in range(D.shape[0]):
            for j in range(i+1, D.shape[1]):
            # check to see if the distance between any two
            # centroid pairs is less than the configured number
            # of pixels
                if D[i, j]<distance_px:
                    # update our violation set with the indexes of
```

```
                    # the centroid pairs
                    # violate.add(i)
                    # violate.add(j)
                    violate[i]=1
                    violate[j]=1

        for i in range(detections):
            x, y = boxes[i][0], boxes[i][1]
            w, h = boxes[i][2], boxes[i][3]
            startX, startY, endX, endY = x,y,x+w,y+h
            color = (0, 255, 0)  # green
            # if the index pair exists within the violation set, then
            # update the color
            # if i in violate: color=(0, 0, 255)  # red
            if violate.get(i) is not None: color=(0, 0, 255)
            # draw (1) a bounding box around the person and (2) the
            # centroid coordinates of the person,
            img = cv.rectangle(img, (startX, startY), (endX, endY), color, 2)
            img = cv.circle(img,(centroids[i][0],centroids[i][1]),1,color,10)
            birds_display = cv.circle(birds_display,(transformed_centroids[i][0],
     transformed_centroids[i][1]),1,color,10)

        # display the rectangle where the bird's magic is happening
        blue = (255,0,0)
        img = cv.line(img,tl,tr,blue,2)
        img = cv.line(img,tl,bl,blue,2)
        img = cv.line(img,bl,br,blue,2)
        img = cv.line(img,tr,br,blue,2)

    # draw the total number of social distancing violations on the output frame
    text = "Social Distancing Violations: {}".format(len(violate))
    img = cv.putText(img, text, (10, img.shape[0]-25), cv.FONT_HERSHEY_SIMPLEX, 0.85, (0, 0,
      255), 3)

    output = np.zeros((400,1282,3),img.dtype)

    img_half = cv.resize(img,(640,360))
    output[0:40,0:640,0:3] = camera_view_heading
    output[40:400,0:640,0:3] = img_half

    birds_display_half = cv.resize(birds_display,(640,360))
    output[0:40,642:1282,0:3] = bird_eye_heading
    output[40:400,642:1282,0:3] = birds_display_half

    out.write(output)

cap.release()
out.release()
print("Processing Completed, Download 'output.mp4' to View Results")
print(f"Time taken to process the input: {time()-initial_time} seconds")
```

Listing 6: Our implementation to process the input video

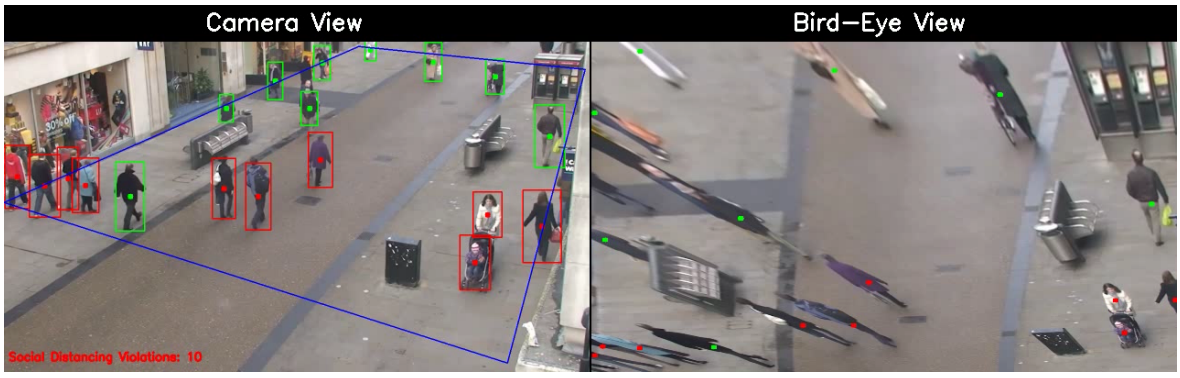


Figure 6: A Still from output video

4.3 The YOLO Algorithm

4.3.1 What is the YOLO Algorithm?

- **YOLO ("You Only Look Once")**[3][4] is an effective real-time **object recognition** algorithm, first described in the seminal 2015 paper by Joseph Redmon et al.
- **Image Classification** done by YOLO algorithm aims at assigning an image to one of a number of different categories (e.g. car, dog, cat, human, etc.), essentially answering the question "What is in this picture?". One image has only one category assigned to it.
- **Object Localization** then allows us to locate our object in the image, so our question changes to "Where is it?".
- **Object Detection** provides the tools for doing just that – finding all the objects in an image and drawing the so-called bounding boxes around them.

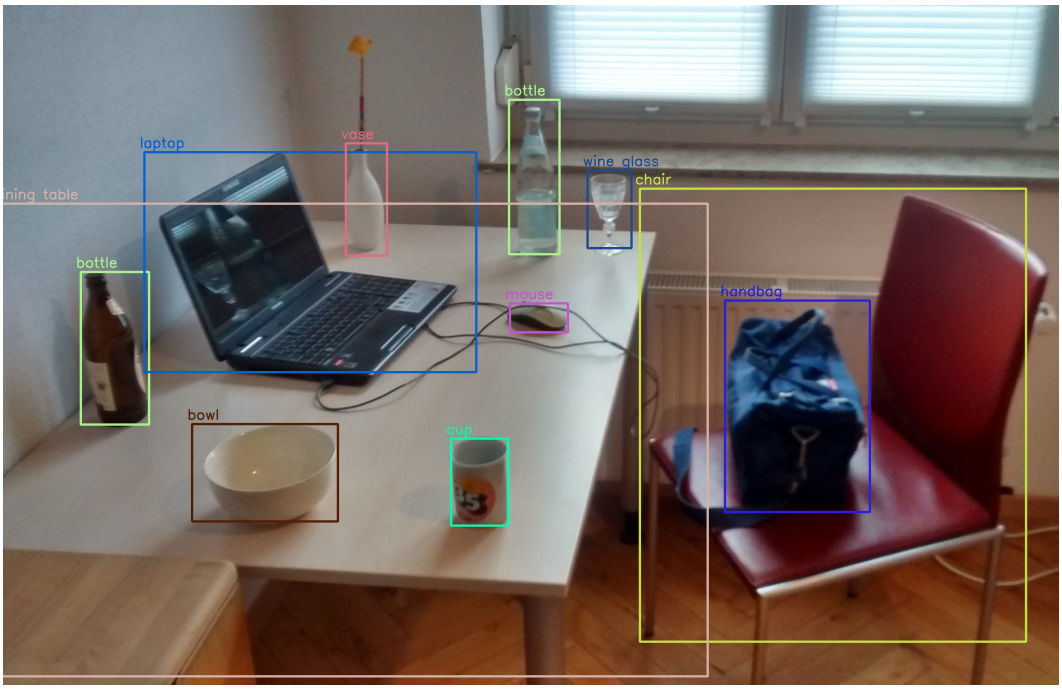


Figure 7: Sample output of YOLOv3

4.3.2 What type of *Object Detection Algorithm* is YOLO?

There are a few different algorithms for object detection and they can be split into two groups:

1. **Algorithms based on Classification**

   They are implemented in two stages. First, they select regions of interest in an image. Second, they classify these regions using convolutional neural networks. This solution can be slow because we have to run predictions for every selected region. A widely known example of this type of algorithm is the Region-based convolutional neural network (RCNN) and its cousins Fast-RCNN, Faster-RCNN and the latest addition to the family: Mask-RCNN. Another example is RetinaNet.

2. **Algorithms based on Regression**

   Instead of selecting interesting parts of an image, these predict classes and bounding boxes for the whole image in one run of the algorithm. The two best known examples from this group are the **YOLO *(it stands here)*** family algorithms and SSD (Single Shot Multibox Detector). They are commonly used for real-time object detection as, in general, they trade a bit of accuracy for large improvements in speed.

4.3.3 How does YOLO work?[5]

To understand the YOLO algorithm, it is necessary to establish what is actually being predicted. Ultimately, we aim to predict a class of an object and the bounding box specifying object location. Each bounding box can be described using four descriptors:

1. Center of the bounding box (b_x, b_y)
2. Width of the bounding box (b_w)
3. Height of the bounding box (b_h)
4. Value corresponding to the class of an object (c=car,person,traffic lights etc)
5. The probability (confidence value) that there is an object bounding the box (p_c)

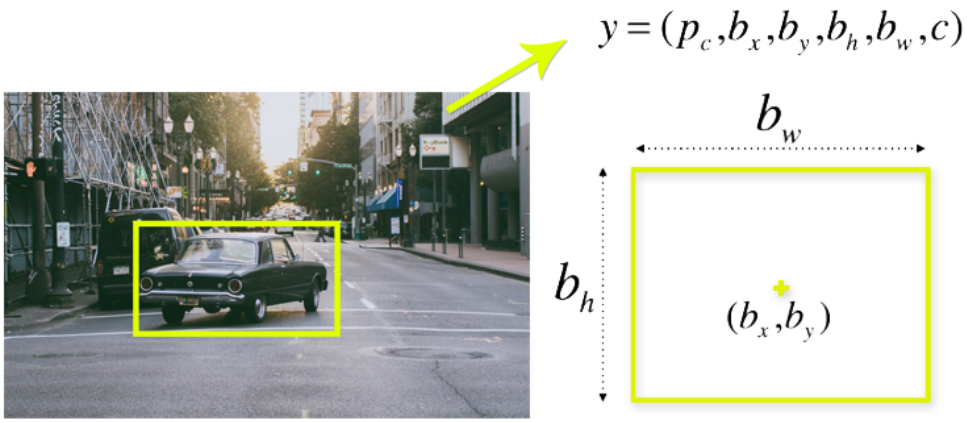


Figure 8: Descriptions of a Bounding Box

Then, the image is split into cells, typically using a 19×19 grid. Each cell is responsible for predicting 5 bounding boxes (in case there are multiple objects in this cell). Therefore, we arrive at a large number of 1805 bounding boxes for one image.

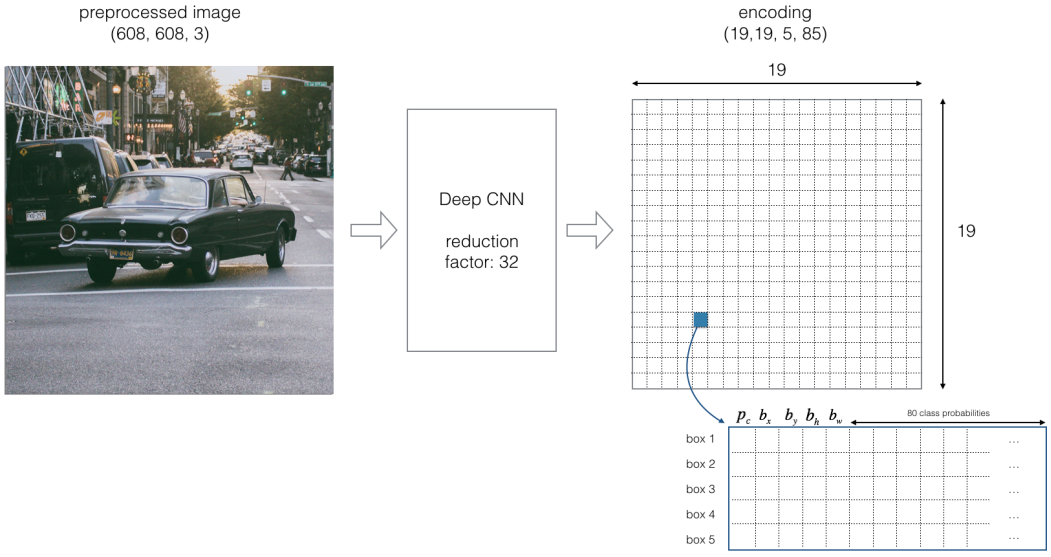


Figure 9: Cell Structure of an Image

Most of these cells and bounding boxes will not contain an object. Therefore, the value p_c is predicted, which serves to remove boxes with low object probability and bounding boxes with the highest shared area in a process called **non-maxima suppression**.

Figure 10: Non-Maxima Suppression

4.4 Darknet implementation of YOLO

There are a few different implementations of the YOLO algorithm on the web. Darknet is one such open source neural network framework. Darknet was written in the C Language and CUDA technology, which makes it really fast and provides for making computations on a GPU, which is essential for real-time predictions.

The Darknet YOLO model that we used here is pre-trained on the COCO (Common Objects in Context) dataset. It can be downloaded here.

5 SHORTCOMINGS

Like every other piece of software, this *checker* is not perfect. It has its own limitations and shortcomings.

1. **The camera that will record the feed needs to be placed at a position high enough** so that the *viewing angle problem* can be avoided. Placing the camera at a horizontal level will not allow the checker to work correctly. For the lowest error margin, the camera needs to be placed perpendicular to the ground, which is not always possible.

2. **Enormous amount of computing power will be needed to make the algorithm work for a live footage**. Even for recorded footage, we were not able to get more than 3-5 frames per second with a decent GPU. This is due to the object detection algorithm taking time in detecting objects. It is not practical to use this *checker* on a live feed.

3. **The minimum social distance needs to be known in pixels beforehand**. This is a lot more difficult than it sounds since a small change in viewing angle can bring a large change in the distance measurements. Plus it is not easy to calculate any distance in pixels. We ourselves have taken arbitrary values here using trial and error to make things work as they should.

4. **The algorithm will completely fail in overly populated areas**. This is due to how YOLO works. It sacrifices accuracy for speed, therefore it really struggles with multiple objects in a single *cell*.

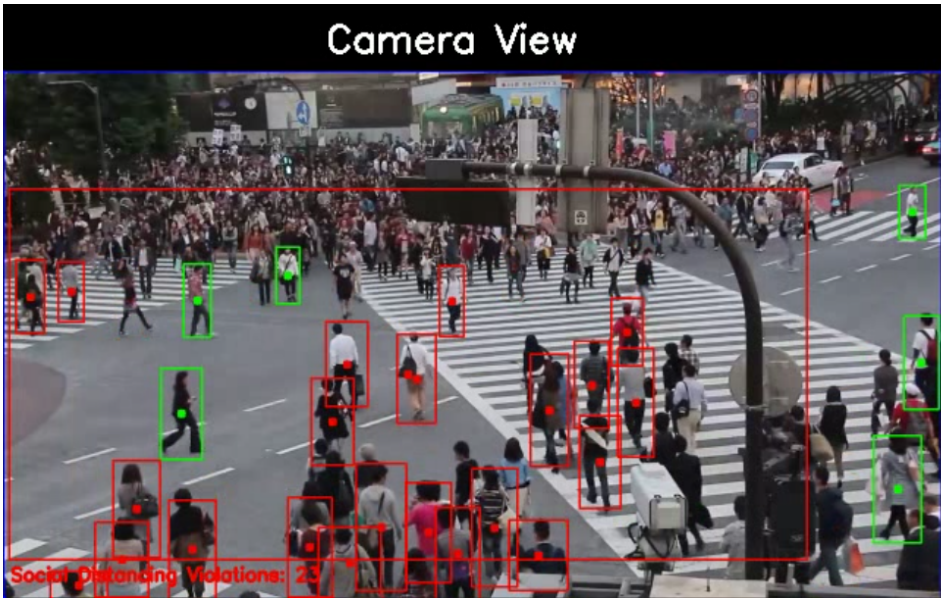


Figure 11: An example of YOLO object detection failing.
The big red box is unintended output.

5.1 Solutions

A few of these limitations can be solved by adopting the following means:

1. Recording via a drone can completely eliminate the *viewing angle problem*, since a drone can be stabilized at exactly 90 degrees to the ground. Indoors, a camera at the center of the ceiling would work wonders.

2. The *checker* can work in densely populated areas as well if we use RCNN or any classification based algorithm. But that will further slow down the *checker* since RCNN is a much slower algorithm than YOLO. So there was something else we tried called **Non-Maxima Suppression analysis**.

3. **Non-Maxima Suppression Analysis**

- To make the algorithm work in relatively dense and overpopulated areas, we were suggested to adjust the Non-Maxima Suppression (NMS) threshold by Prof. Samit Biswas.
- So we decided to try various different values of the NMS threshold. A few of the terms that we used in our NMS analysis are:
  - **Bad Frame**: This is a frame in which the algorithm fails to correctly identify people and instead gives a horrible big box as the output (as shown above).
  - **Total Frames**: This is the number of total frames in the entire video or the length of the video to be analysed.
  - **Performance Ratio**: This is simply (Number of bad frames)/(Total frames).
  - **Object Threshold**: This is the confidence value (p_c) for which a detection is actually considered. Any detection with confidence equal to or above this value is taken into consideration.
- From this it was clear that the threshold value for which the *Performance Ratio* is the lowest would be the suitable and desired value. We also tinkered with the Object Threshold to get the best possible outcome. The script that we wrote for this was:

```
file_name = "shibuya_100frames.mp4"

object_thresholds = [0.25,0.5,0.75,0.85,0.95]
nms_thresholds = np.linspace(0.01,0.025, 30)

plt.figure("NMS vs Performance",figsize=(15,15))
plt.xlabel("NMS Threshold")
plt.ylabel("Performace Ratio ( Bad Frames / Total Frames)")
value = 1

for object_threshold in object_thresholds:
    performance_ratios = []
    for nms_threshold in nms_thresholds:
        clear_output()
        print(f"{value}/{5*30}")
        performance_ratios.append(performance_ratio(object_threshold,nms_threshold,file_name
    ))
        value = value+1
    plt.plot(nms_thresholds,performance_ratios,linestyle = '-',marker = 'o',label = f"
    Obj_Thr = {object_threshold}")

plt.legend(loc = "best")
plt.savefig("NMS vs Performance")
plt.show()
```

Listing 7: NMS Analysis for *"shibuya_100frames.mp4"*

- After running the test for a number of threshold values, we got this graph:

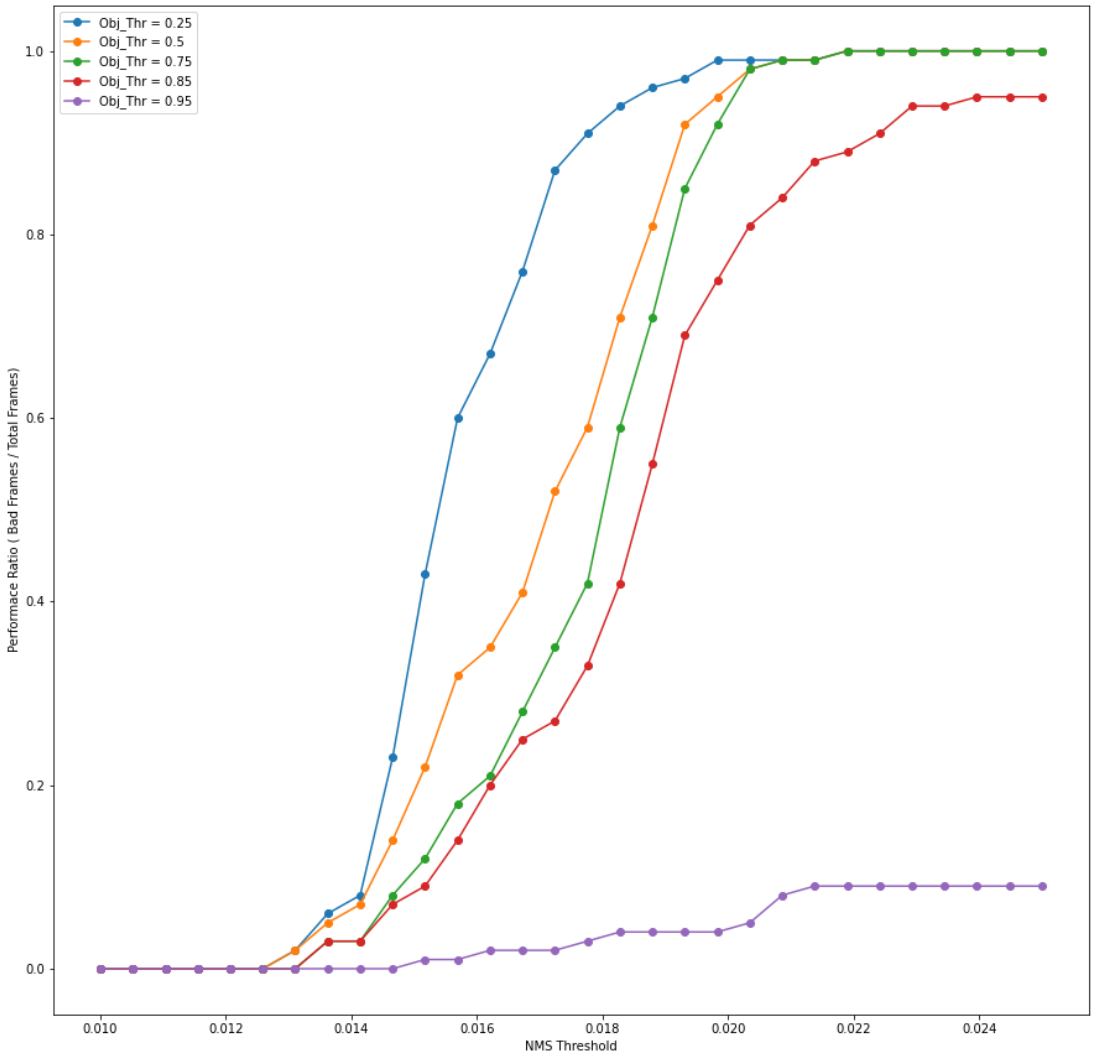


Figure 12: NMS vs. Performance Ratio

- From this, it is visible that the algorithm gives the best results for NMS threshold around 0.010 to 0.014 and for the object threshold greater than 0.9.
- Of course, this is not ideal since **it will ignore most of the detections**.
- So we found a sweet spot at NMS threshold as 0.012 and object threshold as 0.75.
- It should be noted that this analysis will be different for different camera feeds.

6 HENCEFORTH

While keeping the limitations in mind, this program does serve well as a starting point for an automated social distancing checker. The tedious process that needed to be done manually can now be done by a software. This is undoubtedly music to the ears of any software developer and enthusiast.

With that being said, here is how we can improve the *checker*:

1. We can make the entire thing command line based so that an average consumer will not have to dig around the code to calibrate the algorithm to his or her needs.
2. We can (and will) train our own model of YOLO that will only be used to detect people. This can tremendously increase the speed and bring down the processing power requirements
3. We can add a help panel for first time users.

7 REFERENCES

[1] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[2] D. Kriegman, "Homography estimation," *Lecture computer vision I, CSE a*, vol. 252, 2007.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016.

[4] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018.

[5] "Yolo algorithm and yolo object detection: An introduction - appsilon: End to end data science solutions," Oct 2020.