# ABSTRACT

Real-time people flow estimation can be very useful to gain insights for many commercial and noncommercial applications. Counting people on streets or at entrances of places is indeed beneficial for security, tracking, and marketing purposes. People counters can be used to monitor occupancy of entire buildings, individual rooms or anything some of the application where you can implement people counters are

• Retail stores and supermarkets

• Higher education

• Corporate workplaces

• Restaurants, hospitality and leisure facilities

• Washrooms

Vision-based people counting systems have wide potential applications including video surveillance and public resources management. Most works in the literature rely on moving object detection and tracking, assuming that all moving objects are people. The objective of this project is to develop a people counter device to count the number of pedestrians walking through a door or corridor through a video or camera. Most of the time, this system is used at the entrance of a building so that the total number of visitors can be recorded. Live stream of visitors flow is streamed on to a web application.

Keywords – people counting, tracking, deep learning.

## TABLE OF CONTENTS

# 1.INTRODUCTION

## 1.1 OVERVIEW

Counting people in visual surveillance is hard and challenging problem. Automatic counting surveillance of individuals publicly areas is vital for safety control. Previously many techniques and methods are proposed. The motivation for such a system is to gather data about how many persons there is inside a building at any given time. This will help the owners to set up fire extinguishing equipment or the size and placement of fire exits. Building owners are required by law to have enough of this equipment based on how many people that can gather inside. Knowing when and how many customers are inside a shopping mall could also help to optimize labor scheduling and monitor the effectiveness of promotional events. Optimization of security measures is also a possible benefit from this, knowing how many security guards should be assigned, and hot-spots inside the mall for them to patrol.

The goal of this continuation is to enhance the accuracy of the People counting system introduced in motivation. To achieve this, a better separation of groups of people is useful. Also classification of moving non-human objects like a trolley to prevent these from being counted. Methods for classification of people and objects have been proposed by other groups and teams before, selected methods will be put to the test to find one that is accurate in classification as well as being able to work in an real-time environment. This limits the complexity of methods for detection, tracking and classification that can be implemented

## 1.2 PURPOSE

The main objective of this project is to develop a people counter device to count the number of pedestrians walking through a door or corridor through a video or camera. Most of the time, this system is used at the entrance of a building so that the total number of visitors can be recorded. Live stream of visitor’s flow is streamed on to a web application.

# 2. LITERATURE SURVEY

## 2.1 Existing problem

Despite the challenges, crowd counting and monitoring remains an active research area in computer vision in recent  years. Numerous approaches have been proposed over the years. It has an obvious extension to surveillance applications due to the potential for improving safety systems. People counting and tracking in public places has emerged as a hot topic with increases in the global population. This is because population increases have led to a major increase in mass events. People have a significant interest in events such a carnival, sports, concerts, and festivals. However, such events require proper safety measures and crowd management. Heavy crowds' entrance and exit areas present a challenge for organizers to ensure safety and business health. People counting and tracking systems are used to estimate crowd density and track human activities in crowded areas. Such systems can also be used to manage transport systems, schedule labor, monitor promotional events, and improve surveillance

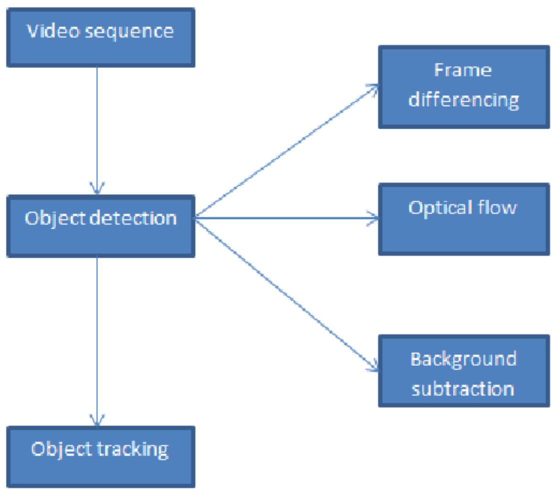
## 2.2 Proposed Solution

Here we are building a model by applying various deep learning algorithms find the best accurate model. Thus we will get to know that people entering and exiting and tracking them. The objective is to build a system that has the following features.

- Read the frames from the Video.
- Draw a desired reference line on the input frame.
- Detect the people using the object detection model.
- Mark the centroid on the detected person.
- Track the movement of that marked centroid.
- Calculate the direction of centroid movement
- Count the number of people coming in or going out of a reference line.
- Based on the counting, increment the up or down counter

# 3.THEORETICAL ANALYSIS:

## 3.1Block Diagram:



## 3.2Hardware / Software Designing:

Software Requirements:

➢ Anaconda Environment

➢ Flask

➢ Python 3.9

➢ And other python libraries like NumPy, pandas.

Hardware Requirements:

• People detection and tracking system

• Create an HTML File.

• Build python code.

• Run the app in local browser.

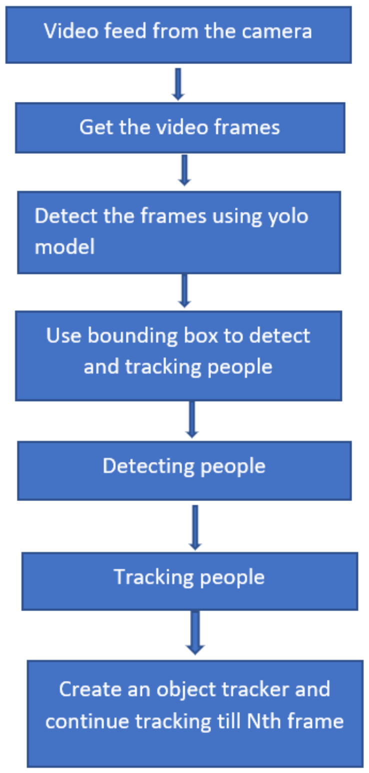
• Show the detection and tracking of people

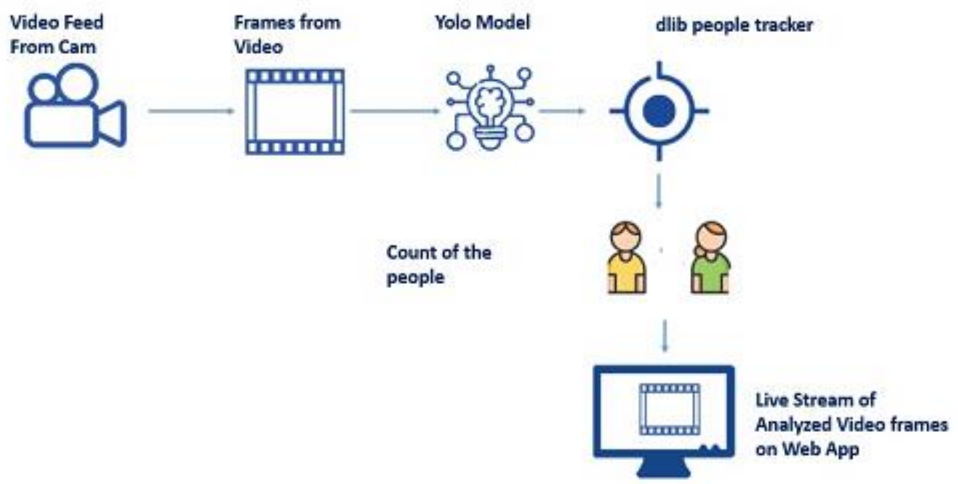
## 4.EXPERIMENTAL INVESTIGATING

The objective of this project is to develop a people counter device to count the number of pedestrians walking through a door or corridor through a video or camera. Most of the time, this system is used at the entrance of a building so that the total number of visitors can be recorded. Live stream of visitors flow is streamed on to a web application.

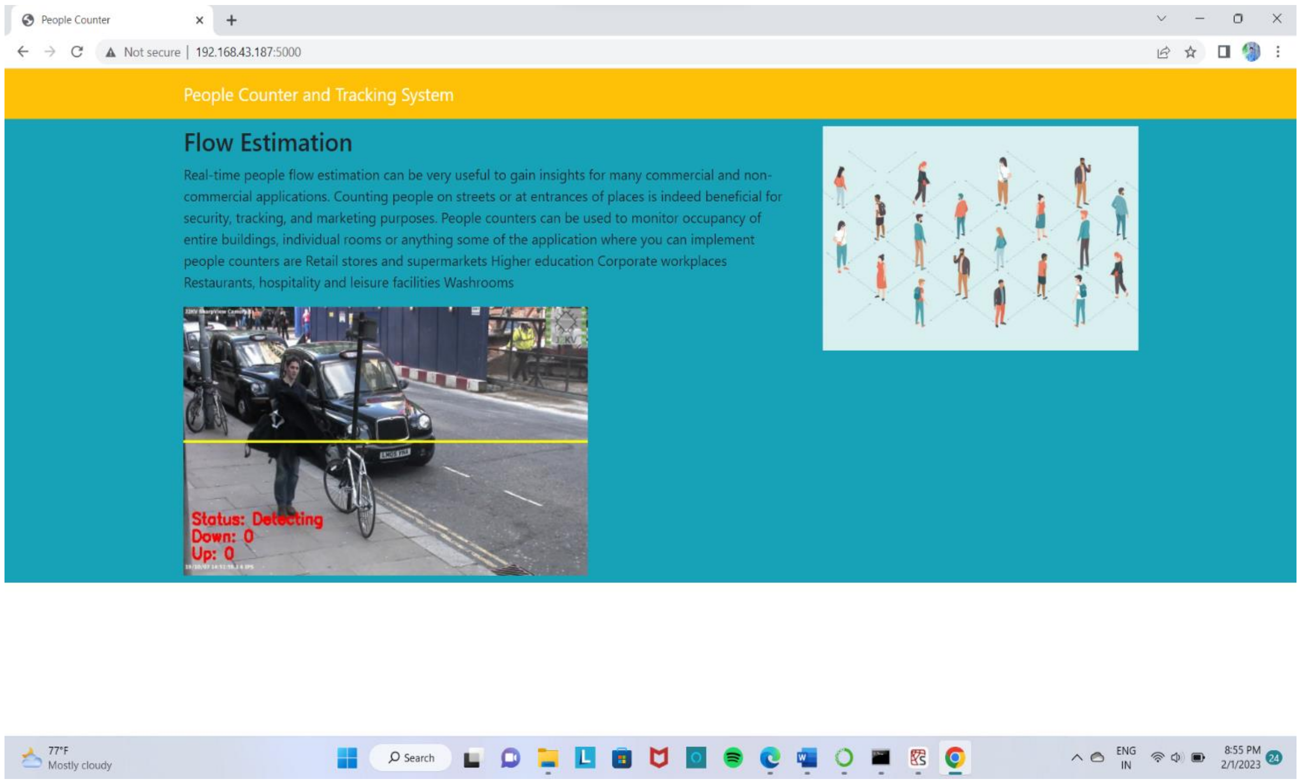
You can collect datasets from different open sources like video frames,computer vision concepts,ccvt cetc. The dataset used for this project was obtained from video frames.

## 5.FLOWCHART:


Video feed from the camera
Get the video frames
Detect the frames using yolo model
Use bounding box to detect and tracking people
Detecting people
Tracking people
Create an object tracker and continue tracking till Nth frame

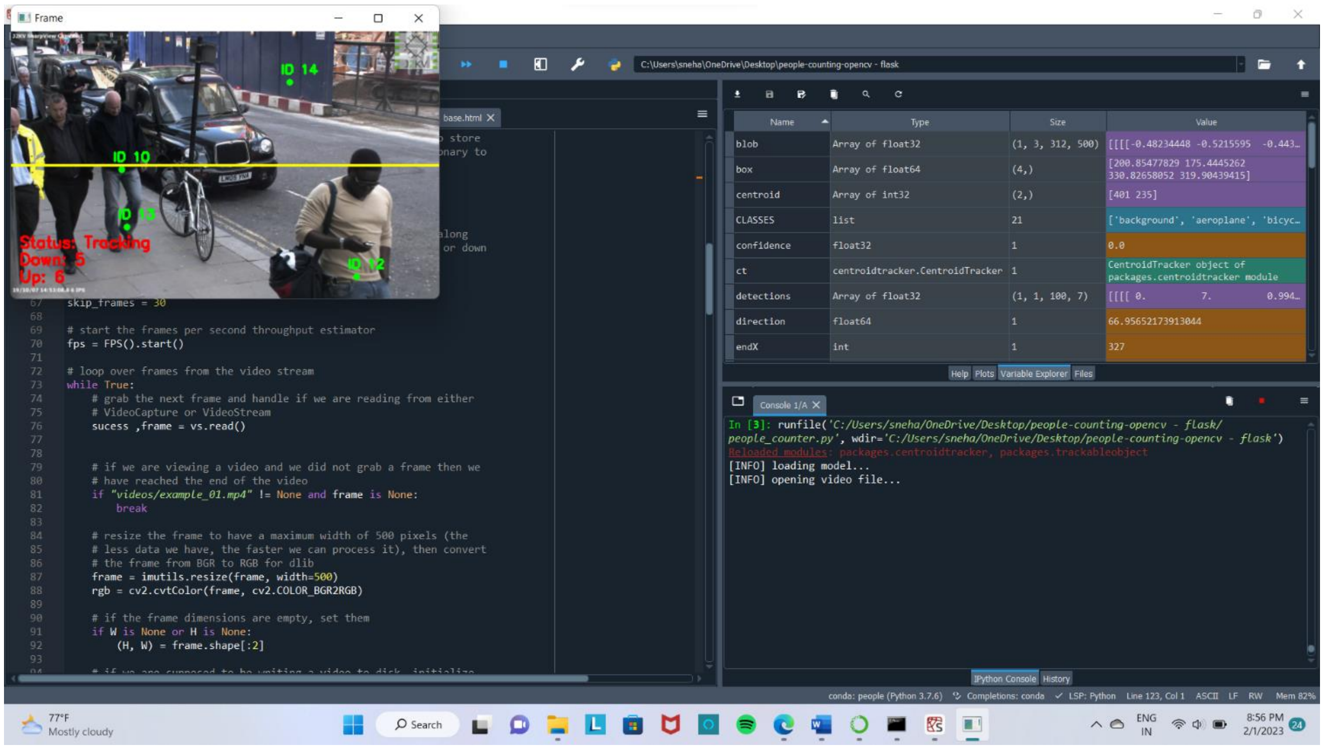


## 6.RESULT



Flow Estimation

Real-time people flow estimation can be very useful to gain insights for many commercial and non-commercial applications. Counting people on streets or at entrances of places is indeed beneficial for security, tracking, and marketing purposes. People counters can be used to monitor occupancy of entire buildings, individual rooms or anything some of the application where you can implement people counters are Retail stores and supermarkets Higher education Corporate workplaces Restaurants, hospitality and leisure facilities Washrooms



```
skip_frames = 30

# start the frames per second throughput estimator
fps = FPS().start()

# loop over frames from the video stream
while True:
    # grab the next frame and handle if we are reading from either
    # VideoCapture or VideoStream
    sucess ,frame = vs.read()


    # if we are viewing a video and we did not grab a frame then we
    # have reached the end of the video
    if "videos/example_01.mp4" != None and frame is None:
        break

    # resize the frame to have a maximum width of 500 pixels (the
    # less data we have, the faster we can process it), then convert
    # the frame from BGR to RGB for dlib
    frame = imutils.resize(frame, width=500)
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # if the frame dimensions are empty, set them
    if W is None or H is None:
        (H, W) = frame.shape[:2]
```

| Name | Type | Size | Value |
|---|---|---|---|
| blob | Array of float32 | (1, 3, 312, 500) | [[[[-0.48234448 -0.5215595 -0.443… |
| box | Array of float64 | (4,) | [200.85477829 175.4445262 330.82658052 319.90439415] |
| centroid | Array of int32 | (2,) | [401 235] |
| CLASSES | list | 21 | ['background', 'aeroplane', 'bicyc… |
| confidence | float32 | 1 | 0.0 |
| ct | centroidtracker.CentroidTracker | 1 | CentroidTracker object of packages.centroidtracker module |
| detections | Array of float32 | (1, 1, 100, 7) | [[[[ 0. 7. 0.994… |
| direction | float64 | 1 | 66.95652173913044 |
| endX | int | 1 | 327 |

```
In [3]: runfile('C:/Users/sneha/OneDrive/Desktop/people-counting-opencv - flask/
people_counter.py', wdir='C:/Users/sneha/OneDrive/Desktop/people-counting-opencv - flask')
Reloaded modules: packages.centroidtracker, packages.trackableobject
[INFO] loading model...
[INFO] opening video file...
```

# 7.ADVANTAGES AND DISADVANTAGES

## ❖ ADVANTAGES:

- ➢ Boost Sales & Conversion
- ➢ Optimize Staffing And Labour Cost
- ➢ Measure Marketing Effectiveness
- ➢ Ensure Social Distancing Practices
- ➢ It provides insight into customer behavior
- ➢ Allows you to understand how external factors affect your business
- ➢ It gives you real time data on usage of space

## ❖ DISADVANTAGES:

- ➢ Accuracy can be affected by the ambient temperature within the counting area being above or below a certain value.
- ➢ Have difficulty measuring the dwell time of people.
- ➢ Needs a camera or other method to verify counts.
- ➢ Technical limitations

## 8.APPLICATIONS:

- Automated public monitoring such as surveillance and traffic control.
- Different from object detection,
- people Counting aims at recognizing crowd in various situations such as public places

## 9.CONCLUSION

The main objective of this project was to develop a system that could perform well in low-to-medium density crowds, with occluded objects and different orientations of the camera. We used the open cv method for human verification. To handle occlusion, multiple particles were distributed on objects verified as humans. Our system is limited in the case of high-density crowds, resulting in reduced accuracy for people detection and tracking

People counters can help to evaluate and compare the number of people visiting stores,and have a wide range of applications.

## 10.FUTURE SCOPE

People counting and tracking system are likely to become more sophisticated and integrated with other technologies such as AI ,Facial recognition and IOT to provide even more valuable insights and improve the over all user experience.some potential applications include

- Retail Analytics
- Public safety
- Workforce management
- Health care
- Smart cities

## 11. BIBLOGRAPHY

## SOURCE CODE:

### Peoplecounter.py:

```
# import the necessary packages
from packages.centroidtracker import CentroidTracker
from packages.trackableobject import TrackableObject
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import dlib
import cv2


# initialize the list of class labels MobileNet SSD was trained to
# detect
CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
	"bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
	"dog", "horse", "motorbike", "person", "pottedplant", "sheep",
	"sofa", "train", "tvmonitor"]

# load our serialized model from disk
print("[INFO] loading model...")
```

```
net = cv2.dnn.readNetFromCaffe(r"modelfiles/MobileNetSSD_deploy.prototxt",
r"modelfiles/MobileNetSSD_deploy.caffemodel")

# if a video path was not supplied, grab a reference to the webcam
if not (r"videos/example_01.mp4", False):
	print("[INFO] starting video stream...")
	vs = VideoStream(src=0).start()
	time.sleep(2.0)

# otherwise, grab a reference to the video file
else:
	print("[INFO] opening video file...")
	vs = cv2.VideoCapture("videos/example_01.mp4")

# initialize the video writer (we'll instantiate later if need be)
writer = None

# initialize the frame dimensions (we'll set them as soon as we read
# the first frame from the video)
W = None
H = None

# instantiate our centroid tracker, then initialize a list to store
# each of our dlib correlation trackers, followed by a dictionary to
# map each unique object ID to a TrackableObject
ct = CentroidTracker(maxDisappeared=40, maxDistance=50)
trackers = []
trackableObjects = {}
```

```
# initialize the total number of frames processed thus far, along
# with the total number of objects that have moved either up or down
totalFrames = 0
totalDown = 0
totalUp = 0
skip_frames = 30

# start the frames per second throughput estimator
fps = FPS().start()

# loop over frames from the video stream
while True:
    # grab the next frame and handle if we are reading from either
    # VideoCapture or VideoStream
    sucess ,frame = vs.read()



    # if we are viewing a video and we did not grab a frame then we
    # have reached the end of the video
    if "videos/example_01.mp4" != None and frame is None:
        break

    # resize the frame to have a maximum width of 500 pixels (the
    # less data we have, the faster we can process it), then convert
    # the frame from BGR to RGB for dlib
    frame = imutils.resize(frame, width=500)
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
```

```
# if the frame dimensions are empty, set them
if W is None or H is None:
	(H, W) = frame.shape[:2]

# if we are supposed to be writing a video to disk, initialize
# the writer
if "output" != None and writer is None:
	fourcc = cv2.VideoWriter_fourcc(*"MJPG")
	writer = cv2.VideoWriter(r"output/output_02.avi", fourcc, 30,
		(W, H), True)

# initialize the current status along with our list of bounding
# box rectangles returned by either (1) our object detector or
# (2) the correlation trackers
status = "Waiting"
rects = []

# check to see if we should run a more computationally expensive
# object detection method to aid our tracker
if totalFrames % skip_frames == 0:
	# set the status and initialize our new set of object trackers
	status = "Detecting"
	trackers = []

	# convert the frame to a blob and pass the blob through the
	# network and obtain the detections
	blob = cv2.dnn.blobFromImage(frame, 0.007843, (W, H), 127.5)
	net.setInput(blob)
```

```
detections = net.forward()

# loop over the detections
for i in np.arange(0, detections.shape[2]):
    # extract the confidence (i.e., probability) associated
    # with the prediction
    confidence = detections[0, 0, i, 2]

    # filter out weak detections by requiring a minimum
    # confidence
    if confidence > 0.4:
        # extract the index of the class label from the
        # detections list
        idx = int(detections[0, 0, i, 1])

        # if the class label is not a person, ignore it
        if CLASSES[idx] != "person":
            continue

        # compute the (x, y)-coordinates of the bounding box
        # for the object
        box = detections[0, 0, i, 3:7] * np.array([W, H, W, H])
        (startX, startY, endX, endY) = box.astype("int")

        # construct a dlib rectangle object from the bounding
        # box coordinates and then start the dlib correlation
        # tracker
        tracker = dlib.correlation_tracker()
```

```
            rect = dlib.rectangle(startX, startY, endX, endY)
            tracker.start_track(rgb, rect)

            # add the tracker to our list of trackers so we can
            # utilize it during skip frames
            trackers.append(tracker)

# otherwise, we should utilize our object *trackers* rather than
# object *detectors* to obtain a higher frame processing throughput
else:
    # loop over the trackers
    for tracker in trackers:
        # set the status of our system to be 'tracking' rather
        # than 'waiting' or 'detecting'
        status = "Tracking"

        # update the tracker and grab the updated position
        tracker.update(rgb)
        pos = tracker.get_position()

        # unpack the position object
        startX = int(pos.left())
        startY = int(pos.top())
        endX = int(pos.right())
        endY = int(pos.bottom())

        # add the bounding box coordinates to the rectangles list
        rects.append((startX, startY, endX, endY))
```

```
# draw a horizontal line in the center of the frame -- once an
# object crosses this line we will determine whether they were
# moving 'up' or 'down'
cv2.line(frame, (0, H // 2), (W, H // 2), (0, 255, 255), 2)

# use the centroid tracker to associate the (1) old object
# centroids with (2) the newly computed object centroids
objects = ct.update(rects)

# loop over the tracked objects
for (objectID, centroid) in objects.items():
	# check to see if a trackable object exists for the current
	# object ID
	to = trackableObjects.get(objectID, None)

	# if there is no existing trackable object, create one
	if to is None:
		to = TrackableObject(objectID, centroid)

	# otherwise, there is a trackable object so we can utilize it
	# to determine direction
	else:
		# the difference between the y-coordinate of the *current*
		# centroid and the mean of *previous* centroids will tell
		# us in which direction the object is moving (negative for
		# 'up' and positive for 'down')
		y = [c[1] for c in to.centroids]
```

```
        direction = centroid[1] - np.mean(y)
        to.centroids.append(centroid)

        # check to see if the object has been counted or not
        if not to.counted:
            # if the direction is negative (indicating the object
            # is moving up) AND the centroid is above the center
            # line, count the object
            if direction < 0 and centroid[1] < H // 2:
                totalUp += 1
                to.counted = True

            # if the direction is positive (indicating the object
            # is moving down) AND the centroid is below the
            # center line, count the object
            elif direction > 0 and centroid[1] > H // 2:
                totalDown += 1
                to.counted = True

    # store the trackable object in our dictionary
    trackableObjects[objectID] = to

    # draw both the ID of the object and the centroid of the
    # object on the output frame
    text = "ID {}".format(objectID)
    cv2.putText(frame, text, (centroid[0] - 10, centroid[1] - 10),
        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
    cv2.circle(frame, (centroid[0], centroid[1]), 4, (0, 255, 0), -1)
```

```
# construct a tuple of information we will be displaying on the
# frame
	info = [
		("Up", totalUp),
		("Down", totalDown),
		("Status", status),
	]

	# loop over the info tuples and draw them on our frame
	for (i, (k, v)) in enumerate(info):
		text = "{}: {}".format(k, v)
		cv2.putText(frame, text, (10, H - ((i * 20) + 20)),
			cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)

	# check to see if we should write the frame to disk
	if writer is not None:
		writer.write(frame)

	# show the output frame
	cv2.imshow("Frame", frame)
	key = cv2.waitKey(1) & 0xFF

	# if the `q` key was pressed, break from the loop
	if key == ord("q"):
		break

	# increment the total number of frames processed thus far and
	# then update the FPS counter
```

```
		totalFrames += 1
		fps.update()

	# stop the timer and display FPS information
	fps.stop()
	print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
	print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

	# check to see if we need to release the video writer pointer
	if writer is not None:
		writer.release()

	# if we are not using a video file, stop the camera video stream
	if not (r"videos/example_02.mp4", False):
		vs.stop()

	# otherwise, release the video file pointer
	else:
		vs.release()

	# close any open windows
	cv2.destroyAllWindows()
```

**Webstreaming.py:**

```
from packages.centroidtracker import CentroidTracker
from packages.trackableobject import TrackableObject
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import dlib
import cv2

from flask import Flask, render_template, Response

app = Flask(__name__)

# initialize the list of class labels MobileNet SSD was trained to
# detect
CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
    "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
    "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
    "sofa", "train", "tvmonitor"]

# load our serialized model from disk
print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(r"modelfiles/MobileNetSSD_deploy.prototxt",
r"modelfiles/MobileNetSSD_deploy.caffemodel")
```

```
# if a video path was not supplied, grab a reference to the webcam
if not (r"videos/example_01.mp4", False):
    print("[INFO] starting video stream...")
    vs = VideoStream(src=0).start()
 # otherwise, grab a reference to the video file
else:
    print("[INFO] opening video file...")
    vs = cv2.VideoCapture("videos/example_01.mp4")

# initialize the video writer (we'll instantiate later if need be)



@app.route('/')
def index():
    return render_template('base.html')


def gen():
    writer = None

    # initialize the frame dimensions (we'll set them as soon as we read
    # the first frame from the video)
    W = None
    H = None

    # instantiate our centroid tracker, then initialize a list to store
    # each of our dlib correlation trackers, followed by a dictionary to
    # map each unique object ID to a TrackableObject
```

```
ct = CentroidTracker(maxDisappeared=40, maxDistance=50)
trackers = []
trackableObjects = {}

# initialize the total number of frames processed thus far, along
# with the total number of objects that have moved either up or down
totalFrames = 0
totalDown = 0
totalUp = 0
skip_frames = 30

# start the frames per second throughput estimator
fps = FPS().start()
while True:
    # grab the next frame and handle if we are reading from either
    # VideoCapture or VideoStream
    sucess ,frame = vs.read()


        # if we are viewing a video and we did not grab a frame then we
    #ave reached the end of the video
    if "videos/example_01.mp4" != None and frame is None:
        print("noframe")
        break
    # resize the frame to have a maximum width of 500 pixels (the
    # less data we have, the faster we can process it), then convert
    # the frame from BGR to RGB for dlib
    frame = imutils.resize(frame, width=500)
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
```

```
# if the frame dimensions are empty, set them
if W is None or H is None:
    (H, W) = frame.shape[:2]
    # if we are supposed to be writing a video to disk, initialize
    # the writer
status = "Waiting"
rects = []
# check to see if we should run a more computationally expensive
# object detection method to aid our tracker
if totalFrames % skip_frames == 0:
    print("detection")
    # set the status and initialize our new set of object trackers
    status = "Detecting"
    trackers = []
    # convert the frame to a blob and pass the blob through the
    # network and obtain the detections
    blob = cv2.dnn.blobFromImage(frame, 0.007843, (W, H), 127.5)
    net.setInput(blob)
    detections = net.forward()
    # loop over the detections
    for i in np.arange(0, detections.shape[2]):
        print("first forloop")
        # extract the confidence (i.e., probability) associated
        # with the prediction
        confidence = detections[0, 0, i, 2]
        # filter out weak detections by requiring a minimum
        # confidence
        if confidence > 0.4:
            print("yes greater")
            # extract the index of the class label from the
            # detections list
            idx = int(detections[0, 0, i, 1])
            # if the class label is not a person, ignore it
            if CLASSES[idx] != "person":
                print("yes person")
                continue
            # compute the (x, y)-coordinates of the bounding box
            # for the object
            box = detections[0, 0, i, 3:7] * np.array([W, H, W, H])
            (startX, startY, endX, endY) = box.astype("int")
            # construct a dlib rectangle object from the bounding
            # box coordinates and then start the dlib correlation
            # tracker
            tracker = dlib.correlation_tracker()
            rect = dlib.rectangle(startX, startY, endX, endY)
            tracker.start_track(rgb, rect)
            # add the tracker to our list of trackers so we can
            # utilize it during skip frames
            trackers.append(tracker)
    # otherwise, we should utilize our object *trackers* rather than
    # object *detectors* to obtain a higher frame processing throughput
else:
    # loop over the trackers
    for tracker in trackers:
        # set the status of our system to be 'tracking' rather
        # than 'waiting' or 'detecting'
        status = "Tracking"
        # update the tracker and grab the updated position
        tracker.update(rgb)
        pos = tracker.get_position()
        # unpack the position object
        startX = int(pos.left())
        startY = int(pos.top())
        endX = int(pos.right())
        endY = int(pos.bottom())
        # add the bounding box coordinates to the rectangles list
        rects.append((startX, startY, endX, endY))
        # draw a horizontal line in the center of the frame -- once an
        # object crosses this line we will determine whether they were
        # moving 'up' or 'down'
cv2.line(frame, (0, H // 2), (W, H // 2), (0, 255, 255), 2)
# use the centroid tracker to associate the (1) old object
# centroids with (2) the newly computed object centroids
objects = ct.update(rects)
# loop over the tracked objects
for (objectID, centroid) in objects.items():
    # check to see if a trackable object exists for the current
    # object ID
    to = trackableObjects.get(objectID, None)
    # if there is no existing trackable object, create one
    if to is None:
        to = TrackableObject(objectID, centroid)
        # otherwise, there is a trackable object so we can utilize it
        # to determine direction
```

```
else:
    # the difference between the y-coordinate of the *current*
    # centroid and the mean of *previous* centroids will tell
    # us in which direction the object is moving (negative for
    # 'up' and positive for 'down')
    y = [c[1] for c in to.centroids]
    direction = centroid[1] - np.mean(y)
    to.centroids.append(centroid)
    # check to see if the object has been counted or not
    if not to.counted:
        # if the direction is negative (indicating the object
        # is moving up) AND the centroid is above the center
        # line, count the object
        if direction < 0 and centroid[1] < H // 2:
            totalUp += 1
            to.counted = True
        # if the direction is positive (indicating the object
        # is moving down) AND the centroid is below the
        # center line, count the object
        elif direction > 0 and centroid[1] > H // 2:
            totalDown += 1
            to.counted = True
# store the trackable object in our dictionary
trackableObjects[objectID] = to
# draw both the ID of the object and the centroid of the
# object on the output frame
text = "ID {}".format(objectID)
cv2.putText(frame, text, (centroid[0] - 10, centroid[1] - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
                cv2.circle(frame, (centroid[0], centroid[1]), 4, (0, 255, 0), -1)
                cv2.imwrite("3.jpg",frame)
            # construct a tuple of information we will be displaying on the
            # frame
            info = [
                ("Up", totalUp),
                ("Down", totalDown),
                ("Status", status),
                ]
            # loop over the info tuples and draw them on our frame
            for (i, (k, v)) in enumerate(info):
                text = "{}: {}".format(k, v)
                cv2.putText(frame, text, (10, H - ((i * 20) + 20)),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)
                # check to see if we should write the frame to disk
            (flag, encodedImage) = cv2.imencode(".jpg", frame)
            yield (b'--frame\r\n' b'Content-Type: image/jpeg\r\n\r\n' +
                      bytearray(encodedImage) + b'\r\n')


@app.route('/video_feed')
def video_feed():
    return Response(gen(),
                    mimetype='multipart/x-mixed-replace; boundary=frame')


if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=False)
    vs.release()
    cv2.destroyAllWindows()
```

**Centroidtracker.py:**

```
# import the necessary packages
from scipy.spatial import distance as dist
from collections import OrderedDict
import numpy as np

class CentroidTracker:
	def __init__(self, maxDisappeared=50, maxDistance=50):
		# initialize the next unique object ID along with two ordered
		# dictionaries used to keep track of mapping a given object
		# ID to its centroid and number of consecutive frames it has
		# been marked as "disappeared", respectively
		self.nextObjectID = 0
		self.objects = OrderedDict()
		self.disappeared = OrderedDict()

		# store the number of maximum consecutive frames a given
		# object is allowed to be marked as "disappeared" until we
		# need to deregister the object from tracking
		self.maxDisappeared = maxDisappeared

		# store the maximum distance between centroids to associate
		# an object -- if the distance is larger than this maximum
		# distance we'll start to mark the object as "disappeared"
		self.maxDistance = maxDistance

	def register(self, centroid):
		# when registering an object we use the next available object
		# ID to store the centroid
		self.objects[self.nextObjectID] = centroid
		self.disappeared[self.nextObjectID] = 0
		self.nextObjectID += 1

	def deregister(self, objectID):
		# to deregister an object ID we delete the object ID from
		# both of our respective dictionaries
		del self.objects[objectID]
		del self.disappeared[objectID]

	def update(self, rects):
		# check to see if the list of input bounding box rectangles
		# is empty
		if len(rects) == 0:
			# loop over any existing tracked objects and mark them
			# as disappeared
			for objectID in list(self.disappeared.keys()):
				self.disappeared[objectID] += 1

				# if we have reached a maximum number of consecutive
				# frames where a given object has been marked as
				# missing, deregister it
				if self.disappeared[objectID] > self.maxDisappeared:
					self.deregister(objectID)

			# return early as there are no centroids or tracking info
```

```
        # to update
        return self.objects

# initialize an array of input centroids for the current frame
inputCentroids = np.zeros((len(rects), 2), dtype="int")

# loop over the bounding box rectangles
for (i, (startX, startY, endX, endY)) in enumerate(rects):
        # use the bounding box coordinates to derive the centroid
        cX = int((startX + endX) / 2.0)
        cY = int((startY + endY) / 2.0)
        inputCentroids[i] = (cX, cY)

# if we are currently not tracking any objects take the input
# centroids and register each of them
if len(self.objects) == 0:
        for i in range(0, len(inputCentroids)):
                self.register(inputCentroids[i])

# otherwise, are are currently tracking objects so we need to
# try to match the input centroids to existing object
# centroids
else:
        # grab the set of object IDs and corresponding centroids
        objectIDs = list(self.objects.keys())
        objectCentroids = list(self.objects.values())

        # compute the distance between each pair of object
# centroids and input centroids, respectively -- our
# goal will be to match an input centroid to an existing
# object centroid
D = dist.cdist(np.array(objectCentroids), inputCentroids)

# in order to perform this matching we must (1) find the
# smallest value in each row and then (2) sort the row
# indexes based on their minimum values so that the row
# with the smallest value as at the *front* of the index
# list
rows = D.min(axis=1).argsort()

# next, we perform a similar process on the columns by
# finding the smallest value in each column and then
# sorting using the previously computed row index list
cols = D.argmin(axis=1)[rows]

# in order to determine if we need to update, register,
# or deregister an object we need to keep track of which
# of the rows and column indexes we have already examined
usedRows = set()
usedCols = set()

# loop over the combination of the (row, column) index
# tuples
for (row, col) in zip(rows, cols):
        # if we have already examined either the row or
        # column value before, ignore it
        if row in usedRows or col in usedCols:
            continue
        # if the distance between centroids is greater than
        # the maximum distance, do not associate the two
        # centroids to the same object
        if D[row, col] > self.maxDistance:
            continue
        # otherwise, grab the object ID for the current row,
        # set its new centroid, and reset the disappeared
        # counter
        objectID = objectIDs[row]
        self.objects[objectID] = inputCentroids[col]
        self.disappeared[objectID] = 0

        # indicate that we have examined each of the row and
        # column indexes, respectively
        usedRows.add(row)
        usedCols.add(col)




    # compute both the row and column index we have NOT yet
    # examined
    unusedRows = set(range(0, D.shape[0])).difference(usedRows)
    unusedCols = set(range(0, D.shape[1])).difference(usedCols)

    # in the event that the number of object centroids is
    # equal or greater than the number of input centroids
```

```
        # we need to check and see if some of these objects have
        # potentially disappeared
        if D.shape[0] >= D.shape[1]:
            # loop over the unused row indexes
            for row in unusedRows:
                # grab the object ID for the corresponding row
                # index and increment the disappeared counter
                objectID = objectIDs[row]
                self.disappeared[objectID] += 1

                # check to see if the number of consecutive
                # frames the object has been marked "disappeared"
                # for warrants deregistering the object
                if self.disappeared[objectID] > self.maxDisappeared:
                    self.deregister(objectID)

        # otherwise, if the number of input centroids is greater
        # than the number of existing object centroids we need to
        # register each new input centroid as a trackable object
        else:
            for col in unusedCols:
                self.register(inputCentroids[col])

    # return the set of trackable objects
    return self.objects
```

**Trackableobject.py**

```
class TrackableObject:
        def __init__(self, objectID, centroid):
                # store the object ID, then initialize a list of centroids
                # using the current centroid
                self.objectID = objectID
                self.centroids = [centroid]

                # initialize a boolean used to indicate if the object has
                # already been counted or not
                self.counted = False
```

**HTML CODE**

```
<html lang="en">

<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>People Counter</title>
   <link href="https://cdn.bootcss.com/bootstrap/4.0.0/css/bootstrap.min.css" rel="stylesheet">
   <script src="https://cdn.bootcss.com/popper.js/1.12.9/umd/popper.min.js"></script>
   <script src="https://cdn.bootcss.com/jquery/3.3.1/jquery.min.js"></script>
   <script src="https://cdn.bootcss.com/bootstrap/4.0.0/js/bootstrap.min.js"></script>
        <style>
        .bg-dark {
                background-color: #42678c!important;
```

```
            }
            #result {
                        color: #0a1c4ed1;
            }
            </style>
</head>

<body>
    <nav class="navbar navbar-dark bg-warning">
        <div class="container">
            <a class="navbar-brand" href="#">People Counter and Tracking System </a>
        </div>
    </nav>
    <div class="navbar navbar-dark bg-info">


                        <div class="container">
                          <div class="row">
                                    <div class="col-sm-8 bd" >
                                      <h3>Flow Estimation</h3>
                                      <p>Real-time people flow estimation can be very useful to gain insights for many commercial and non-commercial applications. Counting people on streets or at entrances of places is indeed beneficial for security, tracking, and marketing purposes. People counters can be used to monitor occupancy of entire buildings, individual rooms or anything some of the application where you can implement people counters are
Retail stores and supermarkets
Higher education
Corporate workplaces
Restaurants, hospitality and leisure facilities
Washrooms</p>
```

```
                        <img src= "{{ url_for('video_feed') }}"
style="height:300px"class="img-rounded" alt="Gesture">

                  </div>

                  <div class="col-sm-4">

                        <div>

                  <img src= "https://www.burohappold.com/wp-
content/uploads/2020/04/social-distancing_gettystock-1024x683.jpg"
style="height:250px"class="img-rounded" alt="Gesture">

                  </div>


             </div>

            </div>

            </div>

  </div>

</body>
```