

## **ABSTRACT**

Through our project introduces a real-time crowd monitoring system equipped with intelligent alert capabilities, employing computer vision techniques notably YOLO (You Only Look Once) object detection to detect and track individuals within crowds. By extracting person bounding box coordinates using YOLO, and employing a centroid tracking algorithm, the system effectively monitors movements across video frames, facilitating crowd counting and dynamic density analysis. Live feedback on crowd presence is provided, accompanied by alerts triggered when crowd density surpasses predetermined thresholds, indicating potential overcrowding situations. Enhanced with multithreading for optimized performance, key features include real-time detection and tracking, dynamic crowd analysis, intelligent alerts, and efficient parallel processing. Designed for versatility, the system finds applicability in diverse settings such as public gatherings, events, and transportation hubs, offering effective crowd management and safety monitoring.

# INTRODUCTION

In contemporary society, the management and monitoring of crowds have become critical concerns across various domains, ranging from public safety and security to transportation management, retail analytics, and event planning. The ability to analyze crowd behavior, detect anomalies, and respond swiftly to potential threats is essential for ensuring the security and efficiency of public spaces and events. However, traditional methods of crowd monitoring, relying on manual observation or static surveillance systems, often fall short in addressing the challenges posed by dynamic and densely populated environments. In response to these limitations, there is a growing demand for advanced technologies capable of automated crowd analysis and anomaly detection in real-time.

This research aims to address this pressing need by developing a real-time crowd monitoring system enhanced with intelligence alert capabilities. Leveraging cutting-edge techniques in computer vision, particularly the You Only Look Once (YOLO) algorithm, and leveraging the richly annotated Common Objects in Context (COCO) dataset, our proposed system seeks to provide efficient, accurate, and proactive crowd monitoring solutions. This introduction provides a comprehensive overview of the motivations driving this research, the objectives guiding its development, and the potential applications of the proposed system.

In modern society, the effective management of crowds has become an increasingly pressing concern across a multitude of sectors and applications. Whether it's ensuring public safety in densely populated urban areas, optimizing traffic flow during major events, or enhancing security measures in public spaces, the ability to monitor and analyze crowd behavior in real-time has become paramount. However, traditional methods of crowd monitoring, which often rely on manual observation or static surveillance systems, are proving inadequate in addressing the complexities of modern environments characterized by rapid urbanization, technological advancements, and evolving security threats.

One of the primary motivations behind this research is the recognition of the limitations inherent in conventional crowd monitoring approaches. Manual surveillance, while useful in certain contexts, is labor-intensive, prone to human error, and lacks scalability when dealing with large crowds or dynamic environments. Static surveillance systems, on the other hand, are limited in their ability to adapt to changing conditions and provide timely insights. As a result, there is a growing demand for automated solutions capable of analyzing vast amounts of visual data in real-time, detecting anomalies, and alerting authorities or stakeholders to potential threats or incidents.

The advent of deep learning-based techniques, particularly in the field of computer vision, has opened up new possibilities for revolutionizing crowd monitoring. The You Only Look Once

(YOLO) algorithm, in particular, has garnered significant attention for its ability to perform real-time object detection with remarkable speed and accuracy. By processing video frames at lightning speed and simultaneously predicting bounding boxes and class probabilities for multiple objects, YOLO has demonstrated its potential to streamline crowd monitoring processes and enable rapid response to emerging situations.

Furthermore, the availability of large-scale annotated datasets such as the Common Objects in Context (COCO) dataset has provided researchers and developers with invaluable resources for training and evaluating deep learning models. The COCO dataset, with its extensive annotations covering a diverse range of object categories, serves as a foundational tool for enhancing the performance and generalization capabilities of object detection algorithms like YOLO. By leveraging the rich annotations and diverse object classes in COCO, researchers can train more robust and accurate models capable of detecting individuals within crowded scenes with greater precision.

Beyond the technical aspects, there is a broader societal imperative driving the need for advanced crowd monitoring solutions. In an era marked by heightened security concerns, urbanization trends, and the proliferation of public events, there is an increasing recognition of the importance of preemptive measures and proactive strategies for maintaining public safety and security. By deploying real-time crowd monitoring systems equipped with intelligence alert capabilities, stakeholders can not only respond more effectively to incidents but also deter potential threats through early detection and intervention.

In summary, the motivation behind this research lies in the recognition of the shortcomings of traditional crowd monitoring methods, the potential of deep learning techniques such as YOLO, and the societal imperative to enhance public safety and security in an increasingly complex and dynamic world. By developing a real-time crowd monitoring system enhanced with intelligence alert capabilities, leveraging the YOLO algorithm and COCO dataset, this research aims to contribute to the advancement of automated crowd analysis and anomaly detection technologies, ultimately fostering safer, more secure, and more resilient.

## **1. Implement and Optimize the YOLO Algorithm for Real-Time Object Detection**

The first objective of this research is to implement and optimize the You Only Look Once (YOLO) algorithm for real-time object detection, with a specific focus on detecting individuals within crowded scenes. This involves understanding the underlying architecture of YOLO, including its convolutional neural network (CNN) backbone, detection head, and loss function. Implementation tasks include adapting the YOLO algorithm to the requirements of crowd monitoring, such as fine-tuning anchor box sizes, adjusting detection thresholds, and optimizing inference speed.

To achieve this objective, rigorous experimentation and optimization strategies will be employed. This includes exploring different YOLO variants (e.g., YOLOv3, YOLOv4) to identify the most suitable architecture for crowd monitoring applications. Additionally, techniques such as model quantization, pruning, and acceleration will be investigated to optimize the inference speed of the YOLO model without sacrificing detection accuracy. The ultimate goal is to develop a highly efficient and accurate object detection framework capable of processing video streams in real-time, thereby enabling timely and proactive crowd monitoring.

## **2. Utilize the COCO Dataset to Train and Fine-Tune the YOLO Model**

The second objective of this research is to leverage the Common Objects in Context (COCO) dataset to train and fine-tune the YOLO model for crowd monitoring applications. The COCO dataset, with its extensive annotations covering a diverse range of object categories, provides a valuable resource for training deep learning models. For the specific task of crowd monitoring, the COCO dataset will be utilized to train the YOLO model to accurately detect and localize individuals within crowded scenes.

To achieve this objective, several steps will be undertaken. First, the COCO dataset will be preprocessed to extract relevant annotations pertaining to individual instances within crowded scenes. Next, data augmentation techniques such as random cropping, rotation, and scaling will be applied to increase the diversity of training samples and improve model generalization. The YOLO model will then be trained using the annotated COCO dataset, with hyperparameters tuned to optimize detection performance. Additionally, transfer learning techniques will be employed to fine-tune the pre-trained YOLO model on crowd-specific datasets, further enhancing its ability to detect individuals in crowded environments.

## **3. Integrate Object Tracking Algorithms for Continuous Monitoring**

The third objective of this research is to integrate object tracking algorithms into the crowd monitoring system to facilitate continuous monitoring and trajectory analysis of detected individuals across consecutive frames. While object detection provides valuable information about the presence and location of individuals within a scene, tracking algorithms are necessary to maintain the identity of detected objects over time and track their movements.

To achieve this objective, various object tracking algorithms will be explored and evaluated, including Kalman filtering, correlation-based tracking, and deep learning-based trackers such as Deep SORT (Simple Online and Realtime Tracking) and SORT (Simple Online and Realtime Tracking). The selected tracking algorithm will be integrated with the

YOLO object detection pipeline, enabling the system to associate detected individuals across frames and estimate their trajectories. This will enable the system to provide not only instantaneous snapshots of crowd dynamics but also insights into long-term trends and patterns.

#### **4. Develop Intelligence Alert Mechanisms**

The fourth objective of this research is to develop intelligence alert mechanisms capable of identifying suspicious behavior, crowd congestion, or other anomalies in real-time based on predefined criteria and thresholds. While object detection and tracking provide valuable information about the location and movement of individuals within a scene, intelligence alerts enable the system to proactively identify potential threats or abnormal situations and alert authorities or stakeholders.

To achieve this objective, heuristic rules, machine learning algorithms, and anomaly detection techniques will be employed to analyze the detected objects and trajectories and identify patterns indicative of suspicious behavior or crowd anomalies. This may include criteria such as sudden changes in crowd density, erratic movement patterns, or loitering behavior in restricted areas. The intelligence alert mechanisms will be designed to provide timely and actionable insights to human operators or decision-makers, enabling them to respond effectively to emerging situations and mitigate potential risks.

## **PROBLEM IDENTIFICATION**

### **The Problem:**

Crowd management and safety monitoring in public spaces, events, and transportation hubs pose significant challenges due to the difficulty in efficiently tracking and analyzing crowd movements. Traditional methods rely on manual surveillance or basic automated systems, often lacking real-time insights and intelligent alert capabilities. This leads to inefficiencies in crowd control, potential safety hazards, and difficulty in responding promptly to overcrowding situations or emergencies.

### **Our Proposed Solution:**

We propose a real-time crowd monitoring system leveraging computer vision techniques, specifically YOLO (You Only Look Once) object detection, and centroid tracking algorithms. By harnessing the capabilities of YOLO, the system detects and tracks individuals within crowds, enabling accurate crowd counting and dynamic density analysis across video frames. The centroid tracking algorithm facilitates continuous monitoring of individual trajectories, allowing for real-time updates on crowd movements.

### **Key Features of Our Solution:**

1. Real-time Detection and Tracking: Utilizing YOLO for efficient detection of individuals in crowded scenes, enabling continuous tracking of their positions.
2. Dynamic Crowd Analysis: Employing centroid tracking to monitor changes in crowd density and movement patterns over time, facilitating dynamic crowd analysis.
3. Intelligent Alerts: Implementing alert mechanisms triggered by predefined thresholds for crowd density, providing timely notifications in case of potential overcrowding situations.
4. Multithreading for Performance Optimization: Enhancing system performance through multithreading, allowing for efficient parallel processing of video frames and reducing processing latency.

### **Application Scenarios:**

1. Public Gatherings: Ensuring crowd safety and efficient management during concerts, festivals, or rallies.
2. Events: Facilitating crowd control and monitoring at sporting events, exhibitions, or conferences.
3. Transportation Hubs: Enhancing safety and efficiency in airports, train stations, or bus terminals by monitoring passenger flows and detecting overcrowding.

## OBJECTIVES

### 1. Real-time Crowd Monitoring and Analysis:

- Utilizing YOLO (You Only Look Once) object detection, the system continuously scans video frames to identify and track individuals within crowds.
- By extracting person bounding box coordinates using YOLO, the system gains insights into the spatial distribution of individuals across frames.
- Employing a centroid tracking algorithm, the system accurately tracks the movements of individuals over successive frames, enabling real-time crowd monitoring.
- Through continuous analysis of crowd dynamics, the system offers insights into crowd behavior, movement patterns, and density fluctuations.

### 2. Crowd Counting and Density Estimation:

- The system performs crowd counting by tallying the number of people entering and exiting predefined areas within the video feed.
- By dynamically updating counts based on individual movements, the system provides real-time crowd density estimates.
- Density analysis enables the identification of areas with high congestion or low occupancy, facilitating proactive crowd management strategies.

### 3. Intelligent Alerting Mechanisms:

- Equipped with intelligent alert capabilities, the system triggers notifications when crowd density surpasses predefined thresholds.
- Alerts serve as early warnings for potential overcrowding situations, enabling timely intervention to ensure public safety and crowd control.

### 4. Efficient Multithreaded Processing:

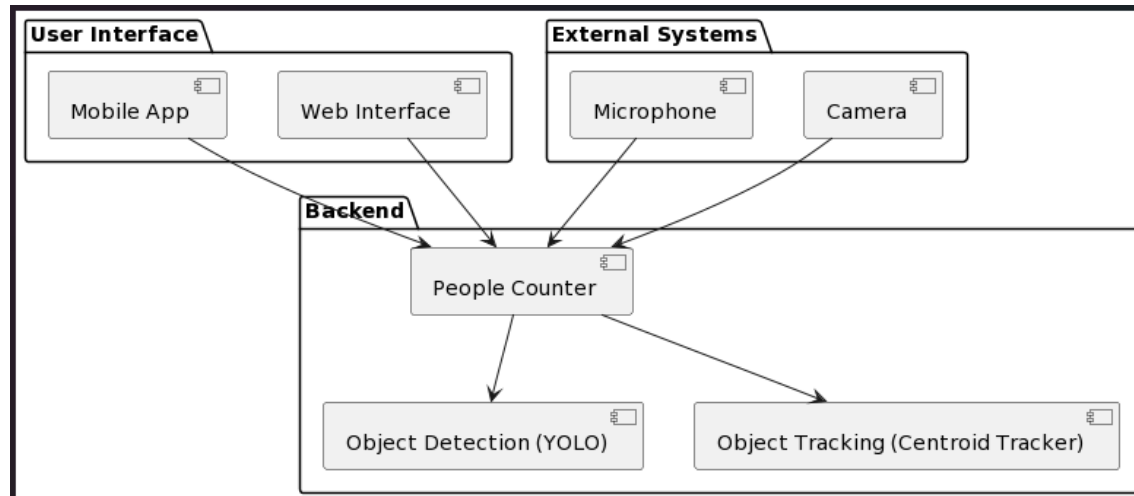
- To achieve real-time performance and optimal resource utilization, the system employs multithreading for parallel processing.
- Multithreading enables concurrent execution of object detection, tracking, crowd counting, and alerting tasks, enhancing overall system efficiency and responsiveness.

### 5. Customization and Adaptability:

- The system offers flexibility and adaptability through parameter customization, allowing users to adjust thresholds, tracking parameters, and alerting criteria to suit specific monitoring scenarios.
- Designed with modularity in mind, the codebase can be extended and integrated with additional functionalities or external systems as per user requirements.

## SYSTEM METHODOLOGY

### Proposed System Architecture:



**Figure 1.** Proposed system Flowchart (Real-Time Crowd Monitoring With Intelligence Alert)

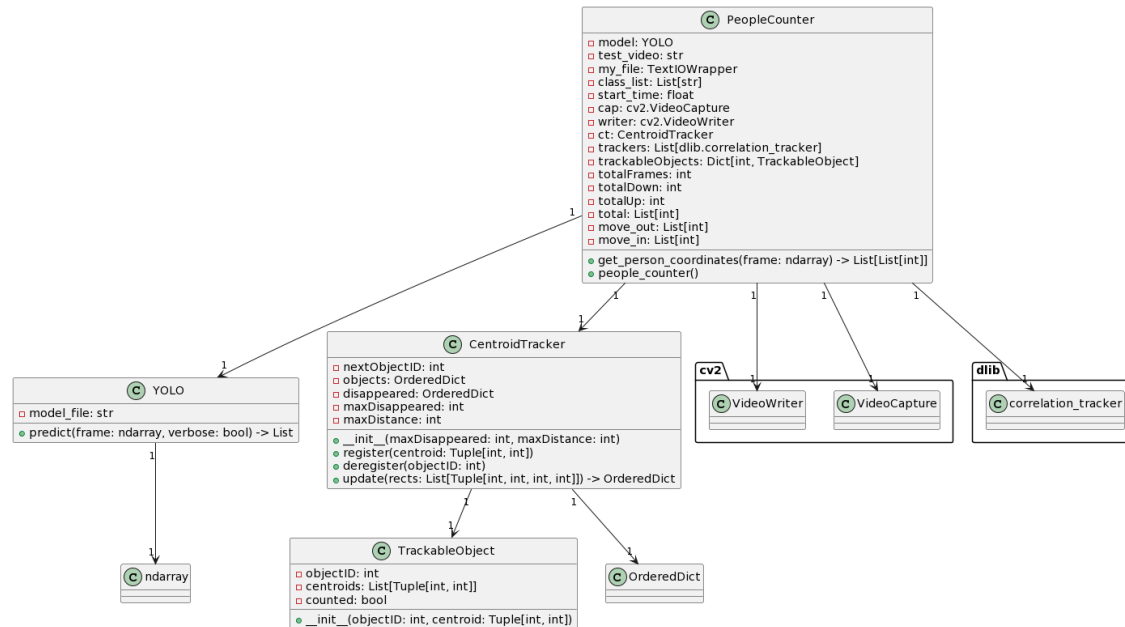
The architecture of the system consists of both hardware and software components. On the hardware side, there are input devices such as a camera, microphone, and speaker. These devices provide the necessary input to the software system.

On the software side, the system is composed of three main components:

1. **PeopleCounter:** This component is responsible for orchestrating the entire system. It interfaces with the hardware components to receive input data and coordinates the processing of this data by the other software components.
2. **YOLO (You Only Look Once):** YOLO is an object detection algorithm used for detecting people in the video feed captured by the camera. It identifies the presence of people in each frame of the video.
3. **CentroidTracker:** The CentroidTracker component is responsible for tracking the movement of people detected by YOLO. It assigns unique IDs to each person and tracks their movement across consecutive frames of the video.

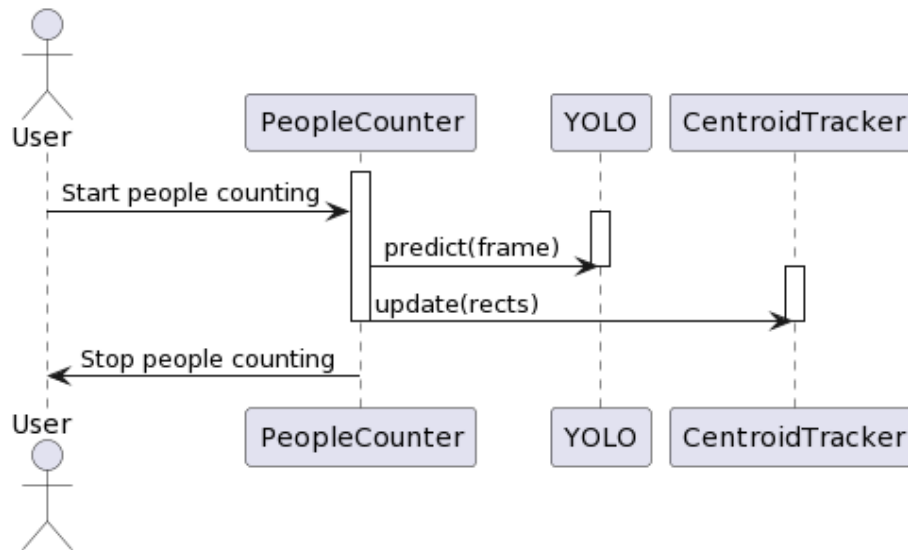


## Class Diagram:



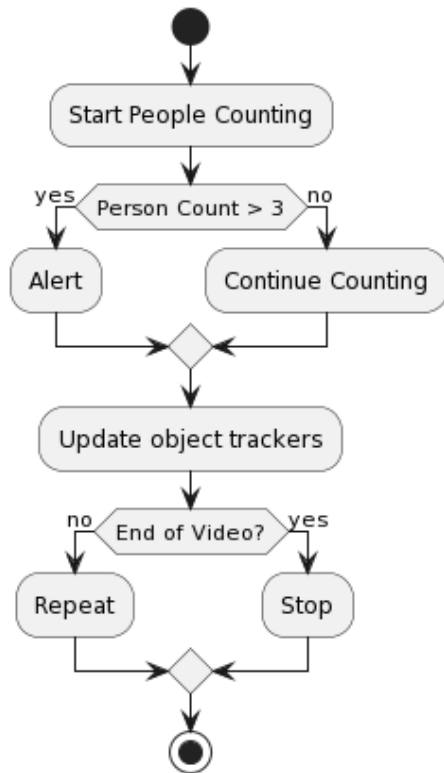
This Class Diagram represents the classes and their relationships in the provided code. You can further extend this diagram or create additional diagrams like Sequence Diagrams, Activity Diagrams, etc., to illustrate different aspects of the system's behavior and structure.

## Sequence Diagram:



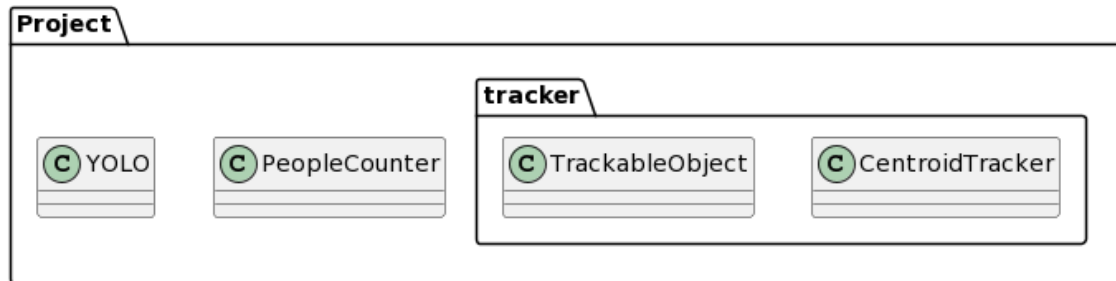
This sequence diagram illustrates the interactions between various components during the people counting process. It shows how the user interacts with the PeopleCounter module, which in turn utilizes the YOLO model for object detection, cv2.VideoCapture for video input, CentroidTracker for object tracking, and cv2.VideoWriter for video output.

### Activity Diagram:



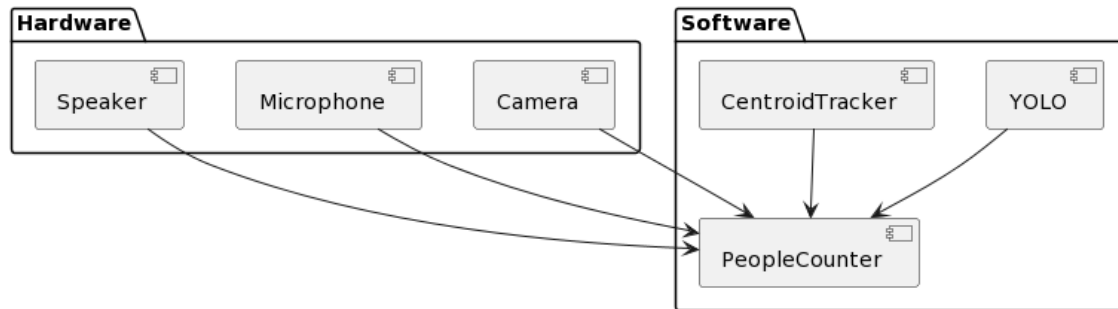
This activity diagram depicts the workflow of the people counting process. It outlines the steps involved, including starting the video processing, processing each frame, detecting people, tracking them, updating the counts, and ending the process when there are no more frames to process.

## Package Diagram:



This package diagram organizes the components of the people counting system into different packages. It shows the division of classes into packages such as "Tracker" for tracking-related modules, "Detector" for detection-related modules, and "Custom Modules" for user-defined modules like PeopleCounter.

## Component Diagram:



This component diagram illustrates the various components of the people counting system and their relationships. It highlights external components like input and output videos, libraries used such as OpenCV, dlib, and the YOLO model, as well as custom modules including the Centroid Tracker, Tracker Objects, and the People Counter module.

## System Requirements

### Hardware Requirements:

- A computer with a GPU (preferably NVIDIA GPU) to run the YOLO model efficiently for real-time object detection.
- Webcam or any other camera device for capturing video input if using live stream.

### Software Requirements:

- Python environment with necessary libraries installed (OpenCV, Pandas, NumPy, Ultralytics, dlib, etc.).
- YOLOv8x model file ('yolov8x.pt') for object detection.
- Pre-trained centroid tracking modules (**CentroidTracker** and **TrackableObject**).

### Implementation:

1. **Code Setup:** Ensure all the necessary libraries are installed and import statements are correctly placed.
2. **Loading YOLO Model:** The YOLO model (YOLOv8x) is loaded using Ultralytics library.
3. **Input Source Selection:** Choose between using a live camera feed or a recorded video. You've provided options for both.
4. **Person Detection:** Utilize the YOLO model to detect people in each frame of the video.
5. **People Counting:** Count the number of people entering and exiting the scene using centroid tracking. This involves associating detected people in each frame with previously detected people and tracking their movement.
6. **Alerting Mechanism:** You've implemented an alerting mechanism to notify if the number of people exceeds a certain threshold.
7. **Video Output:** Optionally, you can save the processed video with counting and tracking information overlaid.

### System Configuration:

- **Hardware:** Ensure your computer meets the hardware requirements, especially if you plan to run the system on a GPU.
- **Software:** Install all required Python libraries and dependencies. You may need to download the YOLOv8x model file ('yolov8x.pt') if not already available.
- **Execution:** Run the main script (**people\_counter()** function) to start the people counting system. Adjust any parameters or thresholds as needed.

### Software Environment:-

#### Python

Python is a versatile, high-level programming language known for its simplicity and readability. Its interpreted nature makes it easy to write and debug code, while its large standard library provides modules and functions for a wide range of tasks. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming, making it suitable for various applications. Additionally, Python is cross-platform, running on Windows, macOS, and Linux, ensuring consistency across different operating systems.

#### Vs code

Visual Studio Code (VS Code) is a lightweight, open-source code editor developed by Microsoft. It offers a wide range of features such as syntax highlighting, code completion, and debugging, making it a popular choice among developers. VS Code's extensive extensions ecosystem allows users to customize their editing environment and add support for different programming languages, frameworks, and tools. With its integrated development environment (IDE) features and cross-platform compatibility, VS Code provides a flexible and efficient environment for coding.

## **Flask**

Flask is a lightweight web application framework for Python, designed to be simple and easy to use. As a microframework, Flask keeps its core minimal and extensible, allowing developers to add functionality as needed. It integrates seamlessly with Jinja2 templating engine for rendering dynamic content in HTML templates and provides a straightforward routing mechanism to handle HTTP requests. Flask is built on top of the Werkzeug WSGI library, which offers utilities for handling HTTP requests and responses, ensuring robust web application development. Overall, Flask is ideal for building small to medium-sized web applications and prototypes, offering flexibility and ease of use.



## OVERVIEW OF TECHNOLOGIES

### **PYTHON - Preferred programming language**

Python is one of the programming languages that is easiest for beginners to learn due of its straightforward syntax.

Python's benefits include:

- Python community that is mature and supportive
- Simple to Learn and Use
- Numerous Python Frameworks and Libraries
- Usage in Big Data, Machine Learning, and Cloud Computing
- Versatility, Efficiency, Reliability, flexibility and Speed

### **Modules in python:**

Certainly, here's an expanded explanation of each imported module:

#### **1. Cv2 (OpenCV):**

OpenCV (Open Source Computer Vision Library) is a widely-used open-source computer vision and machine learning software library. It offers extensive support for image and video processing tasks such as object detection, image filtering, feature detection, and more. OpenCV is renowned for its efficiency and versatility, making it a go-to choice for developers working on computer vision projects across various domains, including robotics, augmented reality, and autonomous vehicles.

## **2. Pandas (pd):**

Pandas is a powerful Python library designed for data manipulation and analysis. It provides easy-to-use data structures like DataFrame and Series, which allow users to efficiently handle structured data. Pandas excels at tasks such as data cleaning, transformation, aggregation, and visualization. With its extensive functionality and seamless integration with other Python libraries, Pandas has become an essential tool for data scientists, analysts, and researchers working with tabular data.

## **3. Numpy (np):**

NumPy is a fundamental Python library for numerical computing. It provides support for multidimensional arrays, matrices, and a wide range of mathematical functions for array manipulation and computation. NumPy's efficient array operations and broadcasting capabilities make it indispensable for scientific computing, machine learning, signal processing, and other numerical tasks. Its seamless integration with other libraries like Pandas and Matplotlib further enhances its usability and versatility.

## **4. Ultralytics.YOLO:**

Ultralytics is a comprehensive library for deep learning and computer vision tasks, offering an easy-to-use interface for running state-of-the-art models. YOLO (You Only Look Once) is a popular object detection algorithm known for its real-time performance and high accuracy. The Ultralytics YOLO module provides pre-trained YOLO models along with utilities for inference, object detection, and evaluation. With Ultralytics YOLO, developers can effortlessly integrate advanced object detection capabilities into their applications.

## **5. Tracker.centroidtracker.CentroidTracker:**

CentroidTracker is a Python class implementing the centroid tracking algorithm for object tracking in video streams. It assigns unique IDs to objects detected in consecutive frames and tracks their movement based on the centroids of their bounding boxes. Centroid tracking is a simple yet effective technique for tracking objects across frames in real-time applications such as video surveillance, traffic monitoring, and human activity recognition.

## **6. Tracker.trackableobject.TrackableObject:**

TrackableObject is a companion class to CentroidTracker, designed to facilitate object tracking by storing essential information about tracked objects. It maintains an object's unique ID, centroid coordinates across frames, and tracking status. TrackableObject plays a crucial role in maintaining object state and ensuring accurate tracking results, making it a vital component of centroid-based tracking systems.

## **7. Imutils.video.FPS:**

FPS (Frames Per Second) is a utility class provided by the imutils library for measuring the frame processing rate of video streams. It calculates the average FPS over a specified number of frames, allowing developers to monitor the performance of their video processing pipeline. FPS measurement is essential for optimizing real-time applications and ensuring smooth video playback, making the FPS class a valuable tool for developers working on video-based projects.

## **8. Dlib:**

Dlib is a modern C++ toolkit containing machine learning algorithms and tools for building complex software solutions. It offers a wide range of functionalities, including face detection, facial landmark detection, object tracking, and more. Dlib's high-performance implementations and easy-to-use APIs make it a popular choice for developers working on computer vision, machine learning, and artificial intelligence projects.

## **9. Logging:**

The logging module in Python provides a flexible framework for emitting log messages from Python programs. It offers support for various logging levels (DEBUG, INFO, WARNING, ERROR, CRITICAL), customizable log message formatting, and configurable logging handlers for outputting log messages to different destinations such as files, streams, or external services. Logging is essential for monitoring program execution, debugging, and troubleshooting issues in production environments.

## **10. Time:**

The time module in Python provides functions for working with time-related tasks, including measuring time durations, working with timestamps, and pausing program execution. It offers functionalities for obtaining the current time, converting between different time representations, and performing time arithmetic. The time module is widely used in applications requiring time-sensitive operations, scheduling, and performance measurement.

## **11. Winsound:**

Winsound is a Python module that provides access to basic sound-playing capabilities on Windows platforms. It allows Python programs to play sound files or system sounds asynchronously, providing developers with a simple interface for incorporating audio feedback into their applications. Winsound is commonly used for signaling events, notifications, or alerts in interactive applications and games running on Windows.

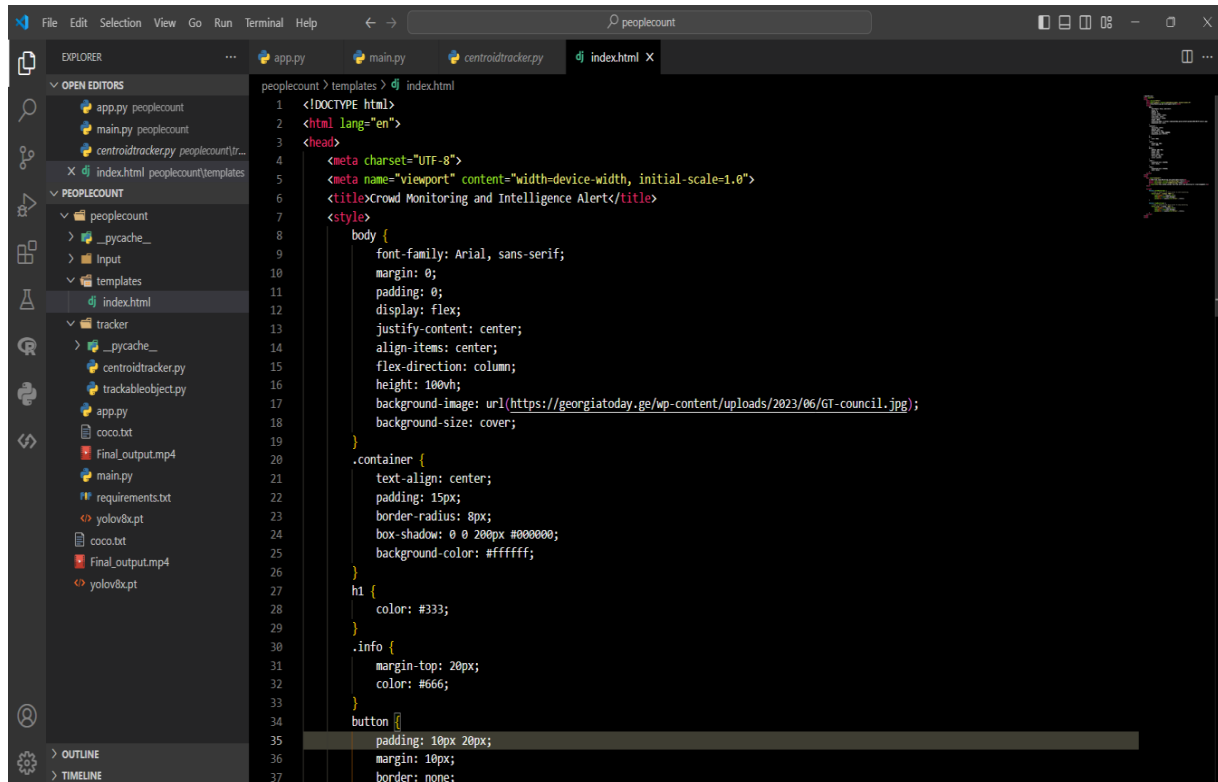
## **12. Threading:**

Threading is a built-in module in Python that provides a high-level interface for working with threads. Threads are lightweight execution units that enable concurrent execution of multiple tasks within a single process. The threading module allows developers to create, start, and synchronize threads, facilitating parallelism and concurrency in Python programs.

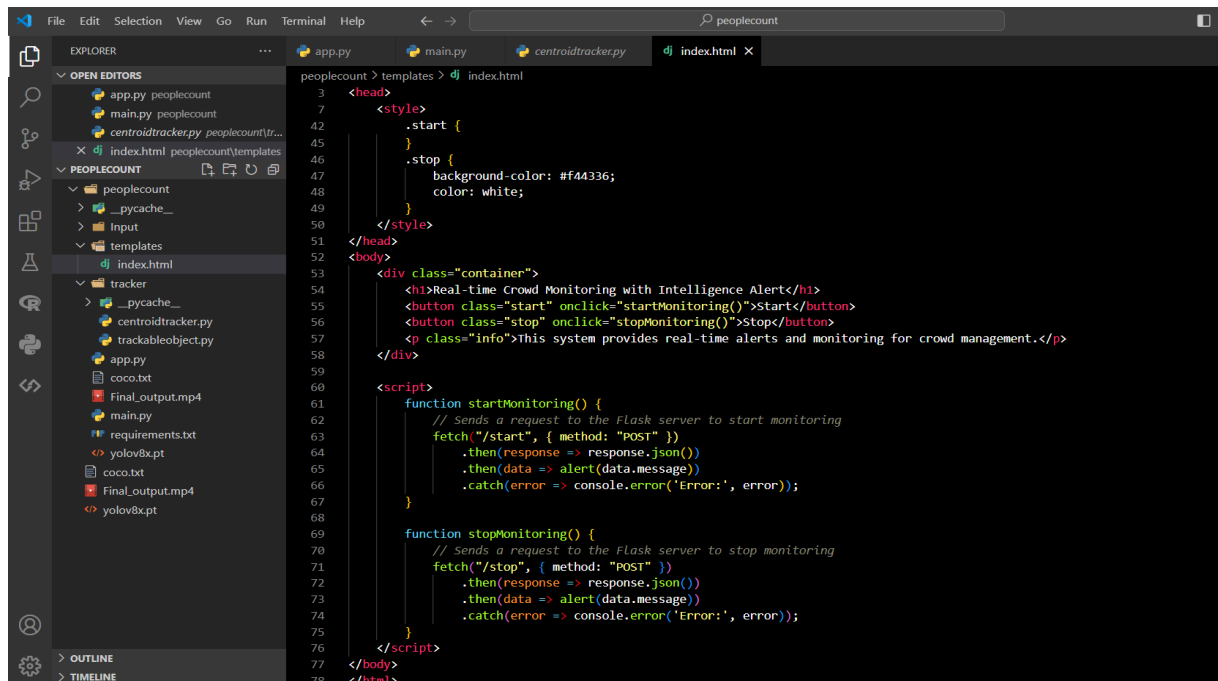
# IMPLEMENTATION

## Code and Tests

### Index.html

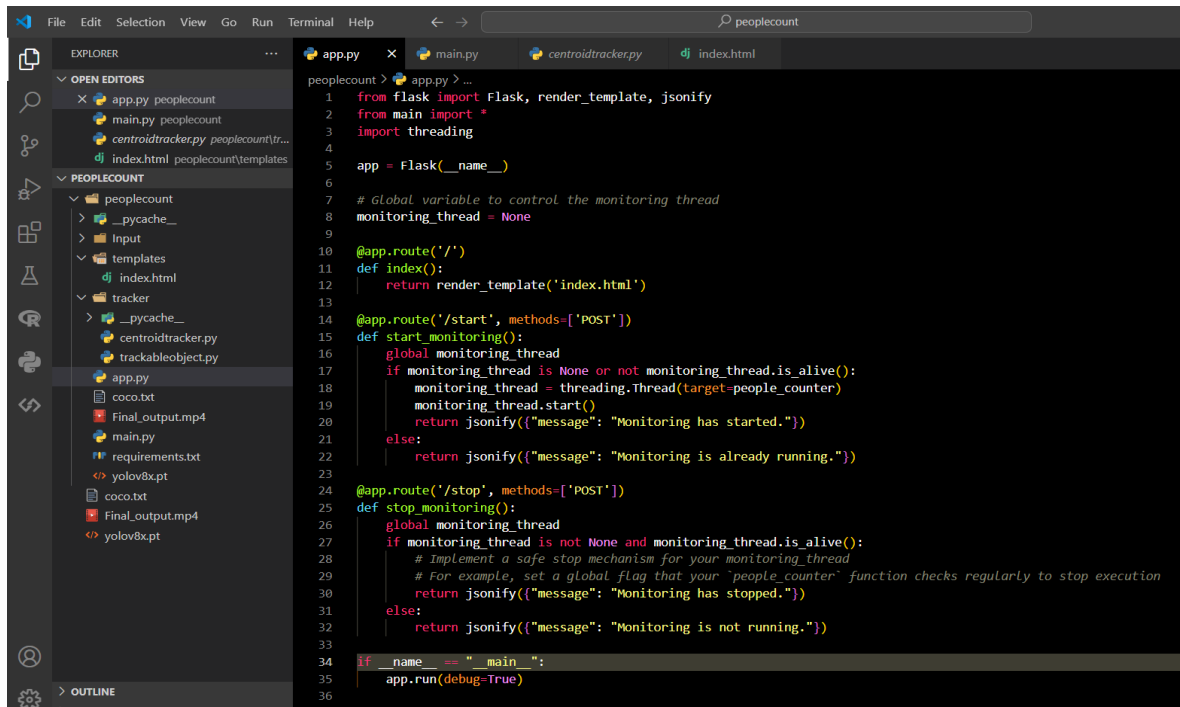


```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Crowd Monitoring and Intelligence Alerts</title>
7 </head>
8 <body>
9   <div>
10     <div>
11       font-family: Arial, sans-serif;
12       margin: 0;
13       padding: 0;
14       display: flex;
15       justify-content: center;
16       align-items: center;
17       flex-direction: column;
18       height: 100vh;
19       background-image: url(https://georgiatoday.ge/wp-content/uploads/2023/06/GT-council.jpg);
20       background-size: cover;
21     </div>
22     .container {
23       text-align: center;
24       padding: 15px;
25       border-radius: 8px;
26       box-shadow: 0 0 20px #000000;
27       background-color: #ffffff;
28     }
29     h1 {
30       color: #333;
31     }
32     .info {
33       margin-top: 20px;
34       color: #666;
35     }
36     button {
37       padding: 10px 20px;
38       margin: 10px;
39       border: none;
```



```
3 <head>
4   <style>
5     .start {
6     }
7     .stop {
8       background-color: #f44336;
9       color: white;
10    }
11  </style>
12 </head>
13 <body>
14   <div class="container">
15     <h1>Real-time Crowd Monitoring with Intelligence Alert</h1>
16     <button class="start" onclick="startMonitoring()">Start</button>
17     <button class="stop" onclick="stopMonitoring()">Stop</button>
18     <p class="info">This system provides real-time alerts and monitoring for crowd management.</p>
19   </div>
20   <script>
21     function startMonitoring() {
22       // Sends a request to the Flask server to start monitoring
23       fetch("/start", { method: "POST" })
24         .then(response => response.json())
25         .then(data => alert(data.message))
26         .catch(error => console.error("Error:", error));
27     }
28     function stopMonitoring() {
29       // Sends a request to the Flask server to stop monitoring
30       fetch("/stop", { method: "POST" })
31         .then(response => response.json())
32         .then(data => alert(data.message))
33         .catch(error => console.error("Error:", error));
34     }
35   </script>
36 </body>
37 </html>
```

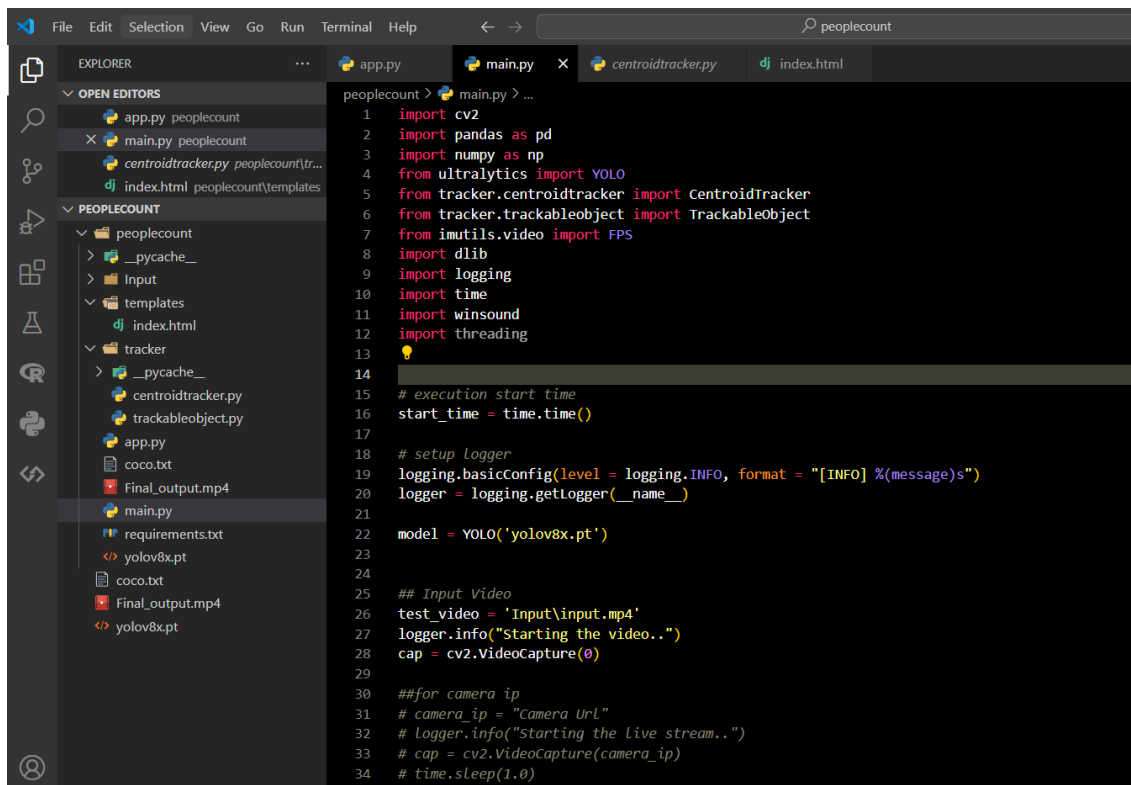
## App.py File



The screenshot shows the Visual Studio Code editor with the 'peoplecount' project open. The Explorer sidebar on the left shows the project structure, including files like app.py, main.py, centroidtracker.py, and index.html. The main editor window displays the code for app.py, which is a Flask application. The code includes imports for Flask, render\_template, jsonify, and threading. It defines a Flask app, a global monitoring thread, and routes for index, start\_monitoring, and stop\_monitoring. The start\_monitoring route starts a thread that monitors people, and the stop\_monitoring route stops it. The app runs in debug mode.

```
1 from flask import Flask, render_template, jsonify
2 from main import *
3 import threading
4
5 app = Flask(__name__)
6
7 # Global variable to control the monitoring thread
8 monitoring_thread = None
9
10 @app.route('/')
11 def index():
12     return render_template('index.html')
13
14 @app.route('/start', methods=['POST'])
15 def start_monitoring():
16     global monitoring_thread
17     if monitoring_thread is None or not monitoring_thread.is_alive():
18         monitoring_thread = threading.Thread(target=people_counter)
19         monitoring_thread.start()
20     return jsonify({"message": "Monitoring has started."})
21 else:
22     return jsonify({"message": "Monitoring is already running."})
23
24 @app.route('/stop', methods=['POST'])
25 def stop_monitoring():
26     global monitoring_thread
27     if monitoring_thread is not None and monitoring_thread.is_alive():
28         # Implement a safe stop mechanism for your monitoring thread
29         # For example, set a global flag that your 'people_counter' function checks regularly to stop execution
30         return jsonify({"message": "Monitoring has stopped."})
31     else:
32         return jsonify({"message": "Monitoring is not running."})
33
34 if __name__ == "__main__":
35     app.run(debug=True)
```

## Main.py File



The screenshot shows the Visual Studio Code editor with the 'peoplecount' project open. The Explorer sidebar on the left shows the project structure, including files like app.py, main.py, centroidtracker.py, and index.html. The main editor window displays the code for main.py, which is a Python script. The code includes imports for cv2, pandas, numpy, ultralytics, tracker, imutils, dlib, logging, and time. It sets up a logger, loads a YOLO model, and starts a video capture. The code is commented to show the execution start time, setup logger, and input video details.

```
1 import cv2
2 import pandas as pd
3 import numpy as np
4 from ultralytics import YOLO
5 from tracker.centroidtracker import CentroidTracker
6 from tracker.trackableobject import TrackableObject
7 from imutils.video import FPS
8 import dlib
9 import logging
10 import time
11 import winsound
12 import threading
13
14 # execution start time
15 start_time = time.time()
16
17 # setup logger
18 logging.basicConfig(level = logging.INFO, format = "[INFO] %(message)s")
19 logger = logging.getLogger(__name__)
20
21 model = YOLO('yolov8x.pt')
22
23 ## Input Video
24 test_video = 'Input\input.mp4'
25 logger.info("Starting the video..")
26 cap = cv2.VideoCapture(0)
27
28 #for camera ip
29 # camera_ip = "Camera Url"
30 # logger.info("Starting the live stream..")
31 # cap = cv2.VideoCapture(camera_ip)
32 # time.sleep(1.0)
```

```
peoplecount > main.py > ...
38 my_file = open("coco.txt", "r")
39 data = my_file.read()
40 class_list = data.split("\n")
41
42
43 #function for detect person coordinate
44 def get_person_coordinates(frame):
45     """
46     Extracts the coordinates of the person bounding boxes from the YOLO model predictions.
47
48     Args:
49         frame: Input frame for object detection.
50
51     Returns:
52         list: List of person bounding box coordinates in the format [x1, y1, x2, y2].
53     """
54     results = model.predict(frame, verbose=False)
55     a = results[0].boxes.data.detach().cpu()
56     px = pd.DataFrame(a).astype("float")
57
58     list_corr = []
59     for index, row in px.iterrows():
60         x1 = row[0]
61         y1 = row[1]
62         x2 = row[2]
63         y2 = row[3]
64         d = int(row[5])
65         c = class_list[d]
66         if 'person' in c:
67             list_corr.append([x1, y1, x2, y2])
68     return list_corr
69
70
```

```
peoplecount > main.py > ...
71 def people_counter():
72     """
73     Counts the number of people entering and exiting based on object tracking.
74     """
75     count = 0
76
77     writer = None
78     ct = CentroidTracker(maxDisappeared=40, maxDistance=40)
79     trackers = []
80     trackableobjects = {}
81
82     # Initialize the total number of frames processed thus far, along
83     # with the total number of objects that have moved either up or down
84     totalFrames = 0
85     totalDown = 0
86     totalUp = 0
87
88     # Initialize empty lists to store the counting data
89     total = []
90     move_out = []
91     move_in = []
92
93     # Initialize video writer
94     W = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
95     H = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
96     fourcc = cv2.VideoWriter_fourcc("mp4v")
97     #if (objectId > 5):
98     writer = cv2.VideoWriter('Final_output.mp4', fourcc, 30, (W, H), True)
99
100     fps = FPS().start()
101     while True:
102         ret, frame = cap.read()
103         if not ret:
104             break
105         count += 1
106         if count % 3 != 0:
107             continue

```

```

71 def people_counter():
109     frame = cv2.resize(frame, (1000, 680))
110
111     per_corr = get_person_coordinates(frame)
112     pcnt = len(per_corr)
113     cv2.putText(frame, str(pcnt), (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1.5, (255, 255, 255), 2)
114
115     if pcnt > 3:
116         cv2.putText(frame, "ALERT!!!", (100, 100), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
117         winsound.PlaySound("SystemExclamation", winsound.SND_ALIAS)
118     else:
119         cv2.putText(frame, " ", (32, 32), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2)
120
121     rects = []
122     if totalFrames % 30 == 0:
123         trackers = []
124         for bbox in per_corr:
125             x1, y1, x2, y2 = bbox
126             rects.append([x1, y1, x2, y2])
127             tracker = dlib.correlation_tracker()
128             rect = dlib.rectangle(int(x1), int(y1), int(x2), int(y2))
129             tracker.start_track(frame, rect)
130             cv2.rectangle(frame, (int(x1), int(y1)), (int(x2), int(y2)), (255, 0, 255), 1)
131             trackers.append(tracker)
132     else:
133         for tracker in trackers:
134             tracker.update(frame)
135             pos = tracker.get_position()
136             startX = int(pos.left())
137             startY = int(pos.top())
138             endX = int(pos.right())
139             endY = int(pos.bottom())
140             rects.append((startX, startY, endX, endY))
141

```

```

144     objects = ct.update(rects)
145
146     for (objectID, centroid) in objects.items():
147         to = trackableObjects.get(objectID)
148
149         if to is None:
150             to = TrackableObject(objectID, centroid)
151         else:
152             y = [c[1] for c in to.centroids]
153             direction = centroid[1] - np.mean(y)
154             to.centroids.append(centroid)
155
156         if not to.counted:
157             if direction < 0 and centroid[1] < H // 2 - 20:
158                 totalUp += 1
159                 move_out.append(totalUp)
160                 to.counted = True
161             elif 0 < direction < 1.1 and centroid[1] > 144:
162                 totalDown += 1
163                 move_in.append(totalDown)
164                 to.counted = True
165
166             total = []
167             total.append(len(move_in) - len(move_out))
168
169         trackableObjects[objectID] = to
170
171         text = "ID {}".format(objectID)
172         #cv2.putText(frame, text, (centroid[0] - 10, centroid[1] - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
173         #            (255, 255, 255), 2)
174         #cv2.circle(frame, (centroid[0], centroid[1]), 4, (255, 255, 255), -1)
175
176     info_status = [
177         ("Enter", totalUp),
178         ("Exit ", totalDown),
179     ]

```



```

71 def people_counter():
181     info_total = [{"Total people inside", ', '.join(map(str, total))}]
182     cntnr=0
183     for (i, (k, v)) in enumerate(info_status):
184         text = "{}: {}".format(k, v)
185         #cv2.putText(frame, text, (10, H - ((i * 20) + 20)), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0), 2)
186         cntnr+=1
187     #cv2.putText(frame, str(cntnr), (50,50), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0), 2)
188     writer.write(frame)
189     cv2.imshow("People Count", frame)
190     cv2.imshow('People Count', frame)
191
192     if cv2.waitKey(1) & 0xFF == 27:
193         break
194
195     totalFrames += 1
196     fps.update()
197
198     end_time = time.time()
199     num_seconds = (end_time - start_time)
200     if num_seconds > 28800:
201         break
202
203     cap.release()
204     writer.release()
205     cv2.destroyAllWindows()
206
207     fps.stop()
208     logger.info("Elapsed time: {:.2f}".format(fps.elapsed()))
209     logger.info("Approx. FPS: {:.2f}".format(fps.fps()))
210
211
212
213 if __name__ == "__main__":
214     people_counter()
215

```

## Centroidtracker.py

```

1 # import the necessary packages
2 from scipy.spatial import distance as dist
3 from collections import OrderedDict
4 import numpy as np
5
6 class CentroidTracker:
7     def __init__(self, maxDisappeared=50, maxDistance=50):
8         # initialize the next unique object ID along with two ordered
9         # dictionaries used to keep track of mapping a given object
10        # ID to its centroid and number of consecutive frames it has
11        # been marked as "disappeared", respectively
12        self.nextObjectID = 0
13        self.objects = OrderedDict()
14        self.disappeared = OrderedDict()
15
16        # store the number of maximum consecutive frames a given
17        # object is allowed to be marked as "disappeared" until we
18        # need to deregister the object from tracking
19        self.maxDisappeared = maxDisappeared
20
21        # store the maximum distance between centroids to associate
22        # an object -- if the distance is larger than this maximum
23        # distance we'll start to mark the object as "disappeared"
24        self.maxDistance = maxDistance
25
26    def register(self, centroid):
27        # when registering an object we use the next available object
28        # ID to store the centroid
29        self.objects[self.nextObjectID] = centroid
30        self.disappeared[self.nextObjectID] = 0
31        self.nextObjectID += 1
32
33    def deregister(self, objectID):
34        # to deregister an object ID we delete the object ID from
35        # both of our respective dictionaries
36        del self.objects[objectID]
37        del self.disappeared[objectID]
38

```

The screenshot shows the VS Code editor with the file explorer on the left and the code editor on the right. The file explorer shows a project structure with folders like 'peoplecount', 'templates', and 'tracker'. The code editor displays the 'register' method of the 'CentroidTracker' class. The method takes a list of rectangles and registers new objects based on their centroids.

```
6 class CentroidTracker:
39
40     def update(self, rects):
41         # check to see if the list of input bounding box rectangles
42         # is empty
43         if len(rects) == 0:
44             # loop over any existing tracked objects and mark them
45             # as disappeared
46             for objectID in list(self.disappeared.keys()):
47                 self.disappeared[objectID] += 1
48
49             # if we have reached a maximum number of consecutive
50             # frames where a given object has been marked as
51             # missing, deregister it
52             if self.disappeared[objectID] > self.maxDisappeared:
53                 self.deregister(objectID)
54
55         # return early as there are no centroids or tracking info
56         # to update
57         return self.objects
58
59     # initialize an array of input centroids for the current frame
60     inputCentroids = np.zeros((len(rects), 2), dtype="int")
61
62     # loop over the bounding box rectangles
63     for (i, (startX, startY, endX, endY)) in enumerate(rects):
64         # use the bounding box coordinates to derive the centroid
65         cX = int((startX + endX) / 2.0)
66         cY = int((startY + endY) / 2.0)
67         inputCentroids[i] = (cX, cY)
68
69     # if we are currently not tracking any objects take the input
70     # centroids and register each of them
71     if len(self.objects) == 0:
72         for i in range(0, len(inputCentroids)):
73             self.register(inputCentroids[i])
```

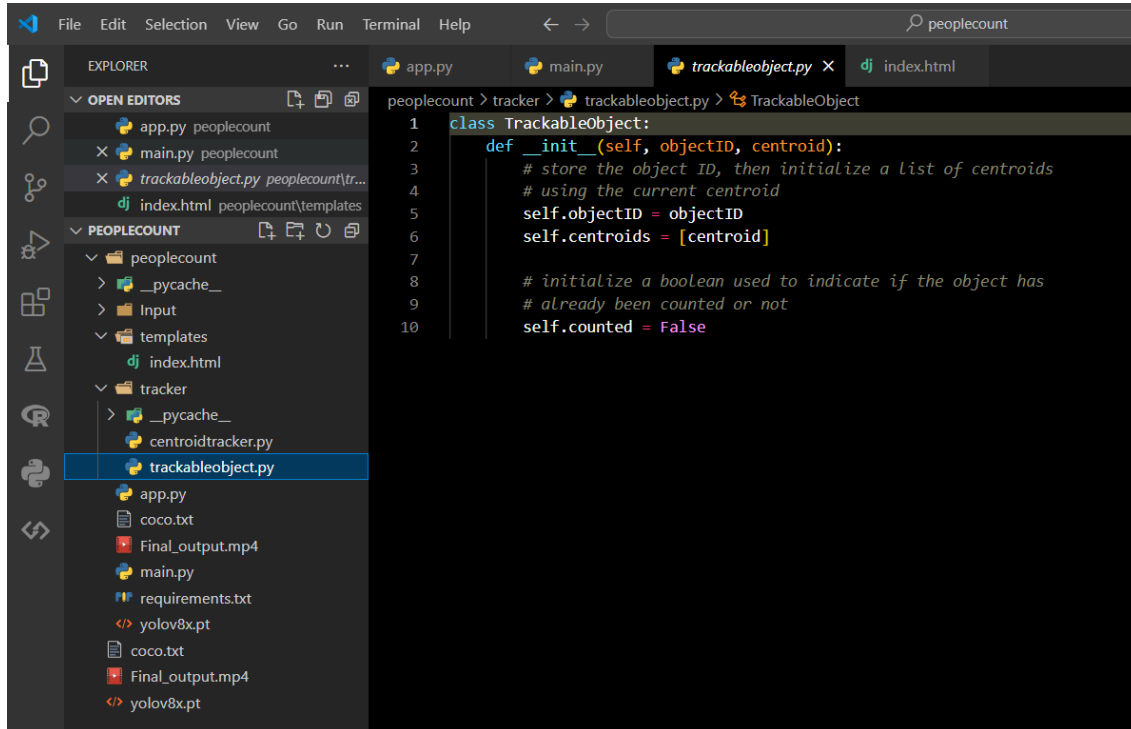
The screenshot shows the VS Code editor with the file explorer on the left and the code editor on the right. The file explorer shows the same project structure as the previous screenshot. The code editor displays the 'update' method of the 'CentroidTracker' class. This method performs a matching process between existing object centroids and new input centroids using distance calculations and sorting.

```
6 class CentroidTracker:
39
77     def update(self, rects):
78         else:
79             # grab the set of object IDs and corresponding centroids
80             objectIDs = list(self.objects.keys())
81             objectCentroids = list(self.objects.values())
82
83             # compute the distance between each pair of object
84             # centroids and input centroids, respectively -- our
85             # goal will be to match an input centroid to an existing
86             # object centroid
87             D = dist.cdist(np.array(objectCentroids), inputCentroids)
88
89             # in order to perform this matching we must (1) find the
90             # smallest value in each row and then (2) sort the row
91             # indexes based on their minimum values so that the row
92             # with the smallest value is at the "front" of the index
93             # list
94             rows = D.min(axis=1).argsort()
95
96             # next, we perform a similar process on the columns by
97             # finding the smallest value in each column and then
98             # sorting using the previously computed row index list
99             cols = D.argmin(axis=1)[rows]
100
101             # in order to determine if we need to update, register,
102             # or deregister an object we need to keep track of which
103             # of the rows and column indexes we have already examined
104             usedRows = set()
105             usedCols = set()
106
107             # loop over the combination of the (row, column) index
108             # tuples
109             for (row, col) in zip(rows, cols):
110                 # if we have already examined either the row or
111                 # column value before, ignore it
112                 if row in usedRows or col in usedCols:
```

```
File Edit Selection View Go Run Terminal Help
peoplecount
EXPLORER
OPEN EDITORS
app.py peoplecount
main.py peoplecount
centroidtracker.py peoplecount\tr...
index.html peoplecount\templates
PEOPLECOUNT
peoplecount
__pycache__
Input
templates
dj index.html
tracker
__pycache__
centroidtracker.py
trackableobject.py
app.py
coco.txt
Final_output.mp4
main.py
requirements.txt
yolov8x.pt
coco.txt
Final_output.mp4
yolov8x.pt
OUTLINE
TIMELINE
peoplecount > tracker > centroidtracker.py > CentroidTracker > update
6 class CentroidTracker:
39 def update(self, rects):
112     continue
113
114     # if the distance between centroids is greater than
115     # the maximum distance, do not associate the two
116     # centroids to the same object
117     if D[row, col] > self.maxDistance:
118         continue
119
120     # otherwise, grab the object ID for the current row,
121     # set its new centroid, and reset the disappeared
122     # counter
123     objectID = objectIDs[row]
124     self.objects[objectID] = inputCentroids[col]
125     self.disappeared[objectID] = 0
126
127     # indicate that we have examined each of the row and
128     # column indexes, respectively
129     usedRows.add(row)
130     usedCols.add(col)
131
132     # compute both the row and column index we have NOT yet
133     # examined
134     unusedRows = set(range(0, D.shape[0])).difference(usedRows)
135     unusedCols = set(range(0, D.shape[1])).difference(usedCols)
136
137     # in the event that the number of object centroids is
138     # equal or greater than the number of input centroids
139     # we need to check and see if some of these objects have
140     # potentially disappeared
141     if D.shape[0] >= D.shape[1]:
142         # Loop over the unused row indexes
143         for row in unusedRows:
144             # grab the object ID for the corresponding row
145             # index and increment the disappeared counter
146             objectID = objectIDs[row]
```

```
File Edit Selection View Go Run Terminal Help
peoplecount
EXPLORER
OPEN EDITORS
app.py peoplecount
main.py peoplecount
centroidtracker.py peoplecount\tr...
index.html peoplecount\templates
PEOPLECOUNT
peoplecount
__pycache__
Input
templates
dj index.html
tracker
__pycache__
centroidtracker.py
trackableobject.py
app.py
coco.txt
Final_output.mp4
main.py
requirements.txt
yolov8x.pt
coco.txt
Final_output.mp4
yolov8x.pt
OUTLINE
TIMELINE
peoplecount > tracker > centroidtracker.py > CentroidTracker > update
6 class CentroidTracker:
39 def update(self, rects):
140     # potentially disappeared
141     if D.shape[0] >= D.shape[1]:
142         # Loop over the unused row indexes
143         for row in unusedRows:
144             # grab the object ID for the corresponding row
145             # index and increment the disappeared counter
146             objectID = objectIDs[row]
147             self.disappeared[objectID] += 1
148
149             # check to see if the number of consecutive
150             # frames the object has been marked "disappeared"
151             # for warrants deregistering the object
152             if self.disappeared[objectID] > self.maxDisappeared:
153                 self.deregister(objectID)
154
155             # otherwise, if the number of input centroids is greater
156             # than the number of existing object centroids we need to
157             # register each new input centroid as a trackable object
158         else:
159             for col in unusedCols:
160                 self.register(inputCentroids[col])
161
162     # return the set of trackable objects
163     return self.objects
```

## Trackableobject.py

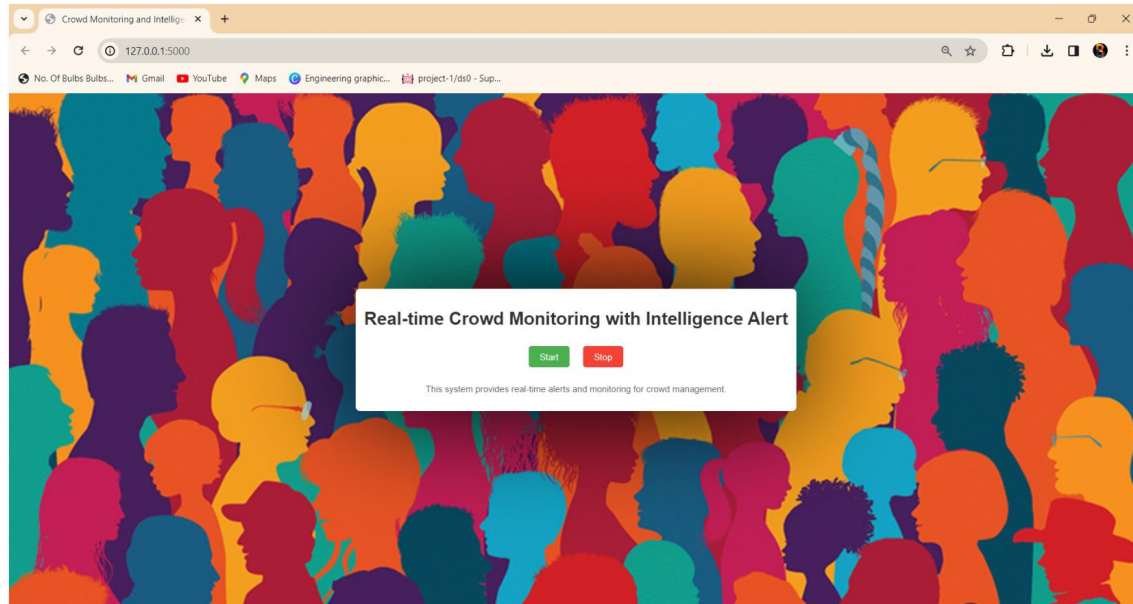


The image shows a Visual Studio Code editor window with the file explorer on the left and the code editor on the right. The file explorer shows a project structure with folders like 'peoplecount', 'templates', and 'tracker'. The file 'trackableobject.py' is selected in the 'tracker' folder. The code editor displays the following Python code:

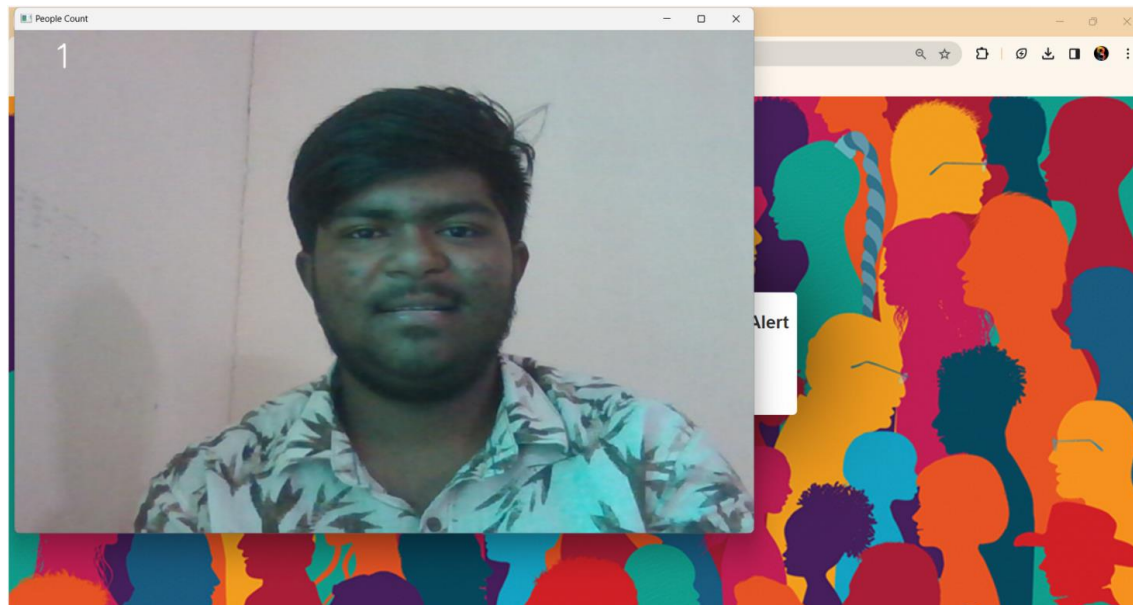
```
1 class TrackableObject:
2     def __init__(self, objectID, centroid):
3         # store the object ID, then initialize a list of centroids
4         # using the current centroid
5         self.objectID = objectID
6         self.centroids = [centroid]
7
8         # initialize a boolean used to indicate if the object has
9         # already been counted or not
10        self.counted = False
```

# RESULTS AND DISCUSSIONS

## WEB INTERFACE

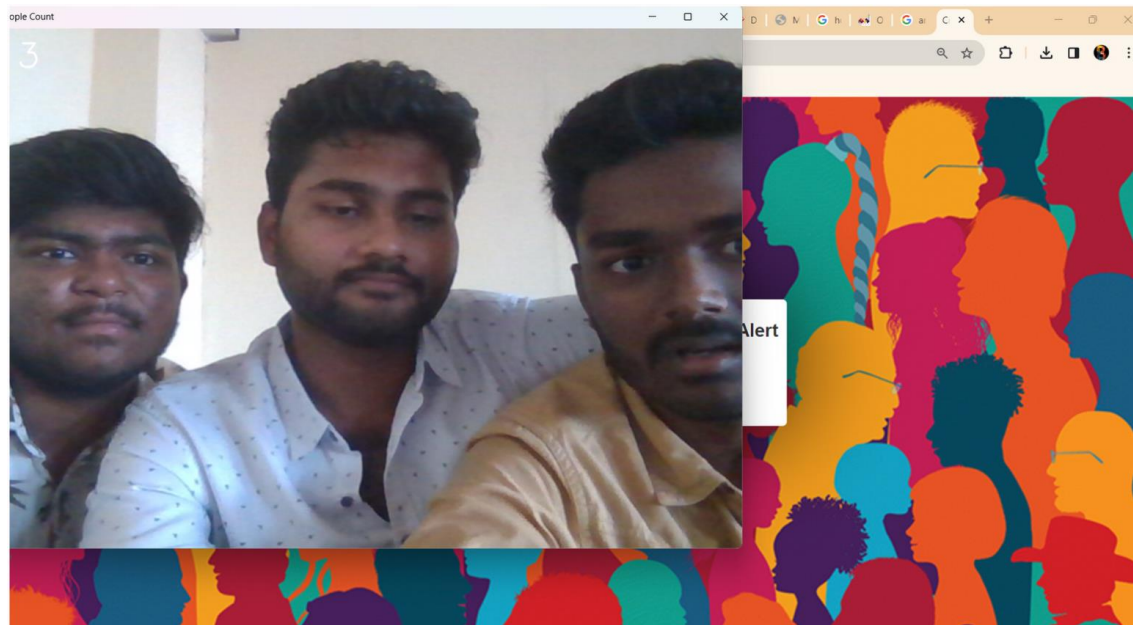


## OUTPUT – 1





## OUTPUT – 2



## OUTPUT – 3



## **CONCLUSION & FUTURE SCOPE**

### **Conclusion:**

The provided code demonstrates a people counter system using YOLO object detection for detecting people in a video stream and centroid tracking for counting the number of people entering and exiting a defined area. The system is capable of detecting and tracking multiple people simultaneously, updating counts accordingly as people enter or exit the scene. It also includes features such as alerting when the number of people exceeds a certain threshold. The code utilizes various libraries and techniques such as OpenCV, YOLO object detection, centroid tracking, and multi-threading to achieve real-time performance. Overall, the implementation showcases a robust framework for monitoring and managing crowd movement in various scenarios.

### **Future Scope:**

There are several avenues for further improvement and expansion of the current system. Firstly, enhancing the accuracy and robustness of person detection and tracking algorithms can lead to more reliable counting results, especially in complex environments with occlusions or crowded scenes. Additionally, integrating deep learning-based re-identification techniques can help improve the tracking accuracy over longer durations and across multiple cameras. Furthermore, the system could be extended to include features such as face recognition for identifying specific individuals or behavior analysis for detecting unusual activities or potential security threats. Moreover, optimizing the code for better performance on resource-constrained devices or integrating it with cloud-based services for scalability and remote monitoring can broaden its applicability. Overall, continuous research and development in computer vision and AI technologies offer numerous opportunities for enhancing the capabilities and applications of people counting systems in various domains such as retail, transportation, security, and smart cities.