Real-Time Pedestrian Tracking and Counting System

with Dynamic Polygon Region-of-Interest Selection

Multi-Object Tracking using YOLOv8 and Centroid Tracking

Dhivya Dharshan V – CS22B2053

Computer Science with Specialization in AI

Indian Institute of Information Technology, Design and Manufacturing Kancheepuram

Pedestrian Tracking System

Real-Time Detection · Multi-Object Tracking Evaluation · Flexible Polygon-based Counting

This report documents implementation, architecture, algorithms, results, and evaluation of a real-time pedestrian tracking and counting system.

November 20, 2025

Abstract

This report presents a comprehensive implementation of a real-time pedestrian tracking and counting system that combines state-of-the-art object detection with robust multi-object tracking algorithms. The system addresses the critical problem of automated people counting in crowded environments, enabling applications in retail analytics, security monitoring, occupancy management, and traffic analysis.

The proposed system integrates YOLOv8 for real-time pedestrian detection with a centroid-based tracking algorithm for temporal consistency. A novel feature of this implementation is the dynamic polygon region-of-interest (ROI) selection, which allows users to define arbitrary 4-point polygonal areas for counting, providing flexibility beyond traditional horizontal line-based counting methods.

The system is evaluated on the MOT17 (Multi-Object Tracking 2017) benchmark dataset using standard MOT metrics including MOTA (Multi-Object Tracking Accuracy), Precision, Recall, and ID switch detection. Results demonstrate the system's effectiveness across different detector variants (DPM, Faster R-CNN, SDP) with average precision of 52.05%, recall of 50.67%, and a total count of 645 pedestrians across 20 MOT17 sequences.

Key contributions include: (1) implementation of interactive polygon ROI selection for flexible area monitoring, (2) robust state-machine-based crossing detection to prevent double-counting, (3) comprehensive MOT evaluation framework with standard metrics, and (4) optimization of YOLO confidence thresholds for crowded scene detection.

Keywords: Multi-Object Tracking, Real-Time Pedestrian Detection, YOLOv8, Centroid Tracking, MOT Challenge, Region-of-Interest Selection, Pedestrian Counting, Computer Vision

MOT17 Dataset Download:
https://motchallenge.net/data/MOT17.zip

Contents

1 Introduction

1.1 Problem Background

Pedestrian counting and tracking in real-world scenarios is a fundamental computer vision problem with extensive applications:

- **Retail Analytics**: Understand customer flow patterns, foot traffic analysis, and store design optimization
- **Security Monitoring**: Automated surveillance, crowd density estimation, anomaly detection
- **Traffic Management**: Vehicle and pedestrian counting at intersections, transportation planning
- **Public Health**: Occupancy management, social distancing monitoring, capacity management
- **Event Management**: Real-time crowd monitoring at events, emergency response coordination

Traditional approaches to pedestrian counting rely on manual observation or simple motion detection, which are labor-intensive, error-prone, and unable to provide temporal tracking information. The need for automated, accurate, and real-time solutions has driven research in multi-object tracking (MOT).

1.2 Motivation and Problem Statement

The core challenges in pedestrian tracking and counting include:

1. **Detection Accuracy**: Detecting all pedestrians in crowded scenes, especially partially occluded or distant individuals
2. **Identity Consistency**: Maintaining consistent object IDs across frames despite occlusions and identity switches
3. **Computational Efficiency**: Achieving real-time performance with limited computational resources
4. **Flexibility**: Adapting to different monitoring scenarios with varying geometry requirements
5. **ROI Definition**: Allowing flexible region-of-interest definition for different use cases (doorways, corridors, retail areas)

PSO - Please Scroll Over

1.3 Objectives

This project aims to:

1. Develop a real-time pedestrian detection and tracking system using modern deep learning
2. Implement interactive polygon ROI selection for flexible area monitoring
3. Provide accurate, directional pedestrian counting (entry/exit detection)
4. Evaluate system performance using MOT17 benchmark with standard metrics
5. Optimize detection and tracking parameters for crowded scenes
6. Create a comprehensive, production-ready implementation

1.4 Report Structure

This report is organized as follows:

- **Section 2**: Literature review and related work
- **Section 3**: System architecture and design
- **Section 4**: Core algorithms and mathematical formulation
- **Section 5**: Implementation details with code examples
- **Section 6**: Experimental setup and methodology
- **Section 7**: Results and analysis
- **Section 8**: Discussion and conclusions
- **Appendix**: Installation guide, project structure, and references

PSO - Please Scroll Over

2 Literature Review and Related Work

2.1 Object Detection

Object detection has evolved significantly with deep learning approaches:

- **R-CNN Family** [1]: Region-based detection with CNN features
- **YOLO** [2]: Single-shot detector enabling real-time detection
- **SSD** [3]: Multi-scale feature maps for detection
- **YOLOv8** [4]: Latest improvements with better accuracy-speed tradeoff

YOLOv8 represents the state-of-the-art in real-time object detection, providing significant improvements in accuracy while maintaining computational efficiency suitable for real-time applications.

2.2 Multi-Object Tracking

Multi-object tracking approaches can be classified into:

2.2.1 Detection-Based Tracking

Detections are generated frame-by-frame and associated across frames using various algorithms:

- **Centroid Tracking**: Simple, fast, based on Euclidean distance between centroids
- **Hungarian Algorithm**: Optimal assignment problem solution
- **Kalman Filtering**: Motion prediction with velocity estimation
- **Deep SORT** [5]: Combines appearance features with motion model

2.2.2 Joint Detection-Tracking

End-to-end approaches that perform detection and tracking simultaneously:

- **FairMOT** [6]: Joint learning for detection and ID features
- **CenterTrack**: Regression-based tracking approach

2.3 MOT Evaluation Metrics

Standard MOT evaluation metrics [7] include:

- **MOTA**: Multi-Object Tracking Accuracy
- **Precision/Recall**: Detection quality metrics
- **ID Switches**: Identity consistency measure
- **MT/ML**: Mostly Tracked/Mostly Lost trajectories
- **IDF1**: Identity F-score [8]

2.4 Related Systems

Existing approaches in pedestrian counting:

- **Static Line Crossing**: Simple horizontal line detection (limited flexibility)
- **Zone-Based Counting**: Multiple fixed zones (configuration complexity)
- **Trajectory Analysis**: Complex motion patterns (high computational cost)

Our system advances the state-of-the-art by combining:

1. State-of-the-art YOLOv8 detector
2. Efficient centroid-based tracking
3. Interactive polygon ROI selection (novel contribution)
4. Comprehensive MOT evaluation framework

PSO - Please Scroll Over

3 System Architecture and Design

3.1 High-Level Architecture

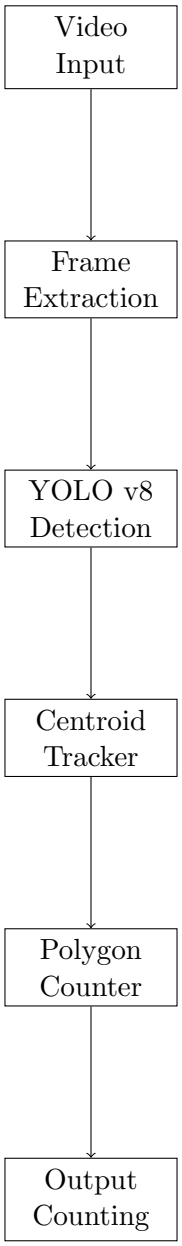


Figure 1: System pipeline overview: from video input to counting output

The system follows a classical detection-tracking-counting pipeline optimized for real-time performance.

PSO - Please Scroll Over

3.2 Module Breakdown

Module	Responsibility	Framework
YOLOv8 Detector	Person detection per frame	Ultralytics
Centroid Tracker	Temporal object association	OpenCV, SciPy
Polygon Counter	ROI crossing detection	OpenCV, NumPy
Main Pipeline	Orchestration and IO	Python

Table 1: System modules and technologies

3.3 Data Flow

1. **Frame Input**: Video frame (typically 1920x1080)
2. **Detection**: YOLOv8 outputs bounding boxes $\{[x_1, y_1, x_2, y_2, conf, cls]\}$
3. **Tracking**: Associate detections to existing tracks using centroid distance
4. **Counting**: Track centroids crossing polygon boundaries
5. **Output**: Directional counts (IN/OUT) and visualization

PSO - Please Scroll Over

4 Core Algorithms and Mathematical Formulation

4.1 YOLOv8 Detection

4.1.1 Architecture Overview

YOLOv8 is a single-stage detector that predicts bounding boxes and class probabilities in one forward pass:

$$\text{Output} = f_{\text{YOLOv8}}(\text{Image}) \tag{1}$$

Where the model outputs:

- Bounding box coordinates: (x, y, w, h) (center and dimensions)
- Confidence score: $p(\text{Object}) \times \text{IoU}(\text{pred}, \text{gt})$
- Class probabilities: $p(\text{class}_i|\text{Object})$

4.1.2 Confidence Filtering

Detections are filtered by confidence threshold:

$$\text{Detection kept if: } p(\text{Object}) \geq \tau_{\text{conf}} \tag{2}$$

Critical insight: For crowded scenes (MOT17), $\tau_{\text{conf}} = 0.3$ outperforms 0.5 as it captures more partially visible pedestrians.

4.2 Centroid-Based Tracking

4.2.1 Centroid Computation

For each bounding box, centroid is computed as:

$$c_x = \frac{x_1 + x_2}{2}, \quad c_y = \frac{y_1 + y_2}{2} \tag{3}$$

4.2.2 Object Association

The tracking algorithm maintains object centroids and matches new detections using Euclidean distance:

$$d_i = \sqrt{(c_{x,det} - c_{x,obj})^2 + (c_{y,det} - c_{y,obj})^2} \tag{4}$$

Matching Rule:

$$\text{Match if: } d_i < d_{\max} \tag{5}$$

PSO - Please Scroll Over

4.2.3 Hungarian Algorithm (Greedy Implementation)

For multiple detections and objects:

1. Compute distance matrix D_{ij} for all detection-object pairs
2. Sort distances in ascending order
3. Greedily assign lowest distances while preventing conflicts
4. Mark unmatched detections as new objects
5. Mark unmatched objects as disappeared

Deregistration:

$$\text{Remove if: disappeared\_frames} > d_{\max} \tag{6}$$

4.2.4 Code: Centroid Tracker Update

```
def update(self, rects):
    # Compute centroids from bounding boxes
    input_centroids = []
    for rect in rects:
        cx = (rect[0] + rect[2]) / 2
        cy = (rect[1] + rect[3]) / 2
        input_centroids.append((cx, cy))

    # If no objects tracked, register all detections
    if len(self.objects) == 0:
        for i, centroid in enumerate(input_centroids):
            self.register(centroid)
        return self.objects

    # Compute distance matrix
    objCentroids = np.array([self.objects[objID]
                             for objID in self.objects])
    inputCentroids = np.array(input_centroids)
    D = distance.cdist(objCentroids, inputCentroids)

    # Greedy matching
    rows = D.min(axis=1).argsort()
    cols = D.argmin(axis=1)[rows]

    usedRows, usedCols = set(), set()
    for (row, col) in zip(rows, cols):
        if D[row, col] < self.max_distance:
            objID = list(self.objects.keys())[row]
            self.objects[objID] = inputCentroids[col]
            self.disappeared[objID] = 0
            usedRows.add(row)
            usedCols.add(col)

```

```
    # Process unmatched objects (disappeared)
    for i in set(range(0, len(objCentroids))) - usedRows:
        objID = list(self.objects.keys())[i]
        self.disappeared[objID] += 1
        if self.disappeared[objID] > self.max_disappeared:
            self.deregister(objID)

    # Process unmatched detections (new objects)
    for col in set(range(0, len(inputCentroids))) - usedCols:
        self.register(inputCentroids[col])

    return self.objects
```

Listing 1: Centroid Tracker Update Loop

4.3 Polygon-Based Boundary Crossing Detection

4.3.1 Point-in-Polygon Test (Ray Casting)

To determine whether a point (x, y) lies inside a polygon, the Ray Casting method is used:

$$\text{count\_intersections} = \text{RayCasting}(\text{point, polygon}) \tag{7}$$

A point is inside the polygon if the number of intersections of the ray is odd:

$$\text{Inside if } (\,\text{count\_intersections} \bmod 2\,) = 1 \tag{8}$$

OpenCV Implementation OpenCV provides an efficient built-in function:

$$\text{result} = \text{cv2.pointPolygonTest(polygon, point, False)} \tag{9}$$

The returned value indicates:

$$\begin{cases} \geq 0 & \text{inside or on boundary} \\ < 0 & \text{outside} \end{cases}$$

4.3.2 Crossing Detection State Machine

Prevent double-counting using state tracking:

$$\text{State}[id] = \{\text{position, last\_position, crossed, distance\_to\_boundary}\} \tag{10}$$

Transition Rule:

$$\text{if } (\text{prev\_pos} \neq \text{curr\_pos}) \wedge \neg\text{crossed then} \begin{cases} \text{count\_in} += 1 & \text{if curr\_pos} = \text{inside} \\ \text{count\_out} += 1 & \text{if curr\_pos} = \text{outside} \end{cases} \tag{11}$$

Reset Condition:

$$\text{if distance\_to\_boundary} > d_{\text{buffer}} \text{ then crossed} = \text{False} \tag{12}$$

4.3.3 Code: Polygon Crossing Detection

```python
def update_and_count(self, objects):
    for object_id, (cx, cy) in objects.items():
        point = (int(cx), int(cy))
        is_inside = self.point_in_polygon(point)

        if object_id not in self.object_states:
            # New object
            position = 'inside' if is_inside else 'outside'
            self.object_states[object_id] = {
                'position': position,
                'last_pos': point,
                'crossed': False
            }
        else:
            state = self.object_states[object_id]
            prev_position = state['position']
            curr_position = 'inside' if is_inside else 'outside'

            # Check crossing
            if prev_position != curr_position and not state['
               crossed']:
                if curr_position == 'inside':
                    self.count_in += 1
                else:
                    self.count_out += 1
                state['crossed'] = True

            # Reset crossed flag if far from boundary
            distance_to_boundary = \
                abs(cv2.pointPolygonTest(
                    self.polygon_points, point, True))
            if distance_to_boundary > self.buffer_distance:
                state['crossed'] = False

            state['position'] = curr_position
            state['last_pos'] = point

    return self.count_in, self.count_out, objects
```

Listing 2: Polygon Boundary Crossing Detection

PSO - Please Scroll Over

4.4 MOT Evaluation Metrics

4.4.1 Intersection over Union (IoU)

Match predictions to ground truth using bounding box overlap:

$$\text{IoU} = \frac{\text{Area(Intersection)}}{\text{Area(Union)}} = \frac{I}{U} \tag{13}$$

Where:

$$I = (x_2 - x_1)(y_2 - y_1) \tag{14}$$

$$U = \text{Area}(\text{Box}_1) + \text{Area}(\text{Box}_2) - I \tag{15}$$

4.4.2 MOTA (Multi-Object Tracking Accuracy)

Standard MOT evaluation metric [7]:

$$\text{MOTA} = 1 - \frac{\sum_t (\text{FP}_t + \text{FN}_t + \text{IDSW}_t)}{\sum_t \text{GT}_t} \tag{16}$$

Where:

- FP_t = False Positives at frame t (detected non-existent objects)
- FN_t = False Negatives at frame t (missed detections)
- IDSW_t = ID Switches at frame t (same object assigned different ID)
- GT_t = Total ground truth detections at frame t

4.4.3 Precision and Recall

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{17}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{18}$$

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{19}$$

4.4.4 ID Switch Detection

Track object-to-prediction mapping across all frames to detect when same GT object gets different predicted ID:

$$\text{ID\_Switch} = |\{t : \text{mapping}[gt\_id](t) \neq \text{mapping}[gt\_id](t-1)\}| \tag{20}$$

PSO - Please Scroll Over

5 Implementation Details

5.1 Core Module: YOLODetector

```python
from ultralytics import YOLO
import numpy as np

class YOLODetector:
    def __init__(self, model_name="yolov8m.pt", conf=0.5, device=0):
        self.model = YOLO(model_name)
        self.conf = conf
        self.device = device

    def detect(self, frame, classes=[0]):
        """Detect persons in frame

        Args:
            frame: Input image (H, W, 3)
            classes: List of class IDs to detect [0=person]

        Returns:
            detections: List of [x1, y1, x2, y2, conf, cls]
        """
        results = self.model(frame, conf=self.conf,
                             device=self.device, verbose=False)

        detections = []
        for result in results:
            for detection in result.boxes:
                # Extract coordinates
                x1, y1, x2, y2 = detection.xyxy[0].cpu().numpy()
                conf = float(detection.conf[0])
                cls_id = int(detection.cls[0])

                # Filter by class
                if cls_id in classes:
                    detections.append([x1, y1, x2, y2, conf,
                                       cls_id])

        return detections
```

Listing 3: YOLOv8 Detector Implementation

Key parameters:

- **model_name**: YOLO model size (n=nano, s=small, m=medium, l=large, x=xlarge)
- **conf**: Confidence threshold (0.3 for crowded scenes recommended)
- **device**: 0 for GPU, 'cpu' for CPU

5.2 Core Module: CentroidTracker

```python
from scipy.spatial import distance
from collections import OrderedDict
import numpy as np

class CentroidTracker:
    def __init__(self, max_disappeared=50, max_distance=50):
        self.next_object_id = 0
        self.objects = OrderedDict()  # {id: (cx, cy)}
        self.disappeared = OrderedDict()  # {id: disappeared\
           _count}
        self.max_disappeared = max_disappeared
        self.max_distance = max_distance

    def register(self, centroid):
        """Register new object"""
        self.objects[self.next_object_id] = centroid
        self.disappeared[self.next_object_id] = 0
        self.next_object_id += 1

    def deregister(self, object_id):
        """Remove object after too long disappeared"""
        del self.objects[object_id]
        del self.disappeared[object_id]
```

Listing 4: Centroid Tracker Class Structure

Parameters:

- **max_disappeared**: Frames before deregistering (50 = 1.6 sec @ 30fps)
- **max_distance**: Max pixel distance for association (50-100 pixels typical)

5.3 Core Module: PolygonCounter

```python
import cv2
import numpy as np

class PolygonCounter:
    def __init__(self, polygon_points, buffer_distance=30):
        self.polygon_points = np.array(polygon_points, dtype=np.
           int32)
        self.object_states = {}
        self.count_in = 0
        self.count_out = 0
        self.buffer_distance = buffer_distance

    def point_in_polygon(self, point):
        """Ray casting algorithm via OpenCV"""
        result = cv2.pointPolygonTest(
            self.polygon_points, point, False)
```

```
        return result >= 0  # True if inside or boundary

    def distance_to_polygon(self, point):
        """Distance to nearest polygon edge"""
        return abs(cv2.pointPolygonTest(
            self.polygon_points, point, True))
```

Listing 5: Polygon Counter Implementation

5.4 Interactive Polygon Selection

```
import cv2

class PolygonSelector:
    def __init__(self, window_name="Select ROI Polygon"):
        self.window_name = window_name
        self.points = []
        self.max_points = 4
        self.image = None
        self.image_copy = None

    def mouse_callback(self, event, x, y, flags, param):
        if event == cv2.EVENT_LBUTTONDOWN:
            if len(self.points) < self.max_points:
                self.points.append((x, y))
                cv2.circle(self.image_copy, (x, y), 5, (0, 255,
                    0), -1)
                cv2.putText(self.image_copy, str(len(self.points)
                    ),
                            (x+10, y+10), cv2.FONT_HERSHEY_SIMPLEX
                                ,
                            0.7, (0, 255, 0), 2)

                if len(self.points) == self.max_points:
                    cv2.line(self.image_copy,
                             self.points[-1], self.points[0],
                             (0, 255, 255), 2)

    def select_polygon(self, frame):
        self.image = frame.copy()
        self.image_copy = frame.copy()
        cv2.namedWindow(self.window_name)
        cv2.setMouseCallback(self.window_name, self.
            mouse_callback)

        while True:
            cv2.imshow(self.window_name, self.image_copy)
            key = cv2.waitKey(1) & 0xFF

            if key == ord('c') and len(self.points) == self.
                max_points:
```

```
                cv2.destroyWindow(self.window_name)
                return self.points
            elif key == 27:  # ESC
                cv2.destroyWindow(self.window_name)
                return None
```

Listing 6: Interactive Polygon Selection Interface

5.5 MOT Evaluation Framework

```
def evaluate_sequence(self, iou_threshold=0.5):
    """Evaluate entire sequence with MOT metrics"""
    gt_to_pred_mapping = {}  # Track GT ID to Pred ID mapping

    for frame_id in sorted(self.ground_truth_tracks.keys()):
        gt_boxes = self.ground_truth_tracks.get(frame_id, [])
        pred_boxes = self.predicted_tracks.get(frame_id, [])

        # Match using IoU
        matches, unmatched_gt, unmatched_pred = \
            self.match_detections(gt_boxes, pred_boxes,
                                  iou_threshold)

        # Update metrics
        self.num_matches += len(matches)
        self.num_misses += len(unmatched_gt)  # FN
        self.num_false_positives += len(unmatched_pred)  # FP

        # ID switch detection
        for gt_id, pred_id in matches:
            if gt_id in gt_to_pred_mapping:
                if gt_to_pred_mapping[gt_id] != pred_id:
                    self.num_switches += 1
            gt_to_pred_mapping[gt_id] = pred_id

    # Calculate final metrics
    num_gt_detections = sum(len(boxes)
                            for boxes in
                            self.ground_truth_tracks.values())

    mota = 1 - (self.num_false_positives + self.num_misses +
                self.num_switches) / num_gt_detections

    precision = self.num_matches / \
        (self.num_matches + self.num_false_positives)
    recall = self.num_matches / \
        (self.num_matches + self.num_misses)

    return {'MOTA': mota*100, 'Precision': precision*100,
            'Recall': recall*100}
```

Listing 7: MOT Evaluation: MOTA Calculation

6 Experimental Setup and Methodology

6.1 Dataset Description

6.1.1 MOT17 Benchmark

The MOT17 (Multi-Object Tracking 2017) dataset is the standard benchmark for evaluating pedestrian tracking systems:

- **Total Sequences**: 21 sequences (14 train, 7 test)
- **Total Frames**: 11,000+ frames
- **Resolution**: 1920×1080 (HD)
- **Frame Rate**: 25-30 FPS
- **Detectors**: Three detector variants (DPM, Faster R-CNN, SDP) per sequence
- **Ground Truth**: Annotated pedestrian bounding boxes with IDs and visibility flags

6.1.2 Sequence Characteristics

Evaluated sequences in this project:

Sequence	Frames	GT IDs	Detections	Characteristics
MOT17-02	600	62	18,581	Crowded street scene
MOT17-04	1050	77	27,072	Busy intersection
MOT17-05	837	133	25,478	Dense pedestrian area
MOT17-09	525	34	9,823	Outdoor plaza
MOT17-10	654	57	11,976	Train station
MOT17-11	900	75	14,370	Shopping center
MOT17-13	750	110	18,090	Market scene

Table 2: MOT17 sequences evaluated

6.2 Experimental Configuration

6.2.1 Detection Parameters

Parameter	Value
Model	YOLOv8m (medium)
Confidence Threshold	0.3
Input Resolution	1920×1080
Device	CPU

Table 3: YOLO Detection Configuration

PSO - Please Scroll Over

6.2.2 Tracking Parameters

Parameter	Value
Max Disappeared Frames	50
Max Distance (pixels)	50
Distance Metric	Euclidean
Matching Algorithm	Greedy (sorted by distance)

Table 4: Centroid Tracker Configuration

6.2.3 MOT Evaluation Parameters

Parameter	Value
IoU Threshold	0.5
Mostly Tracked (MT)	$\geq 80\%$ coverage
Mostly Lost (ML)	$\leq 20\%$ coverage
Partially Tracked (PT)	$20 - 80\%$ coverage

Table 5: MOT Evaluation Configuration

6.3 Metrics and Evaluation Protocol

Primary Metrics:

1. MOTA: Overall multi-object tracking accuracy
2. Precision: Detection quality (TP/(TP+FP))
3. Recall: Detection coverage (TP/(TP+FN))

Secondary Metrics:

1. ID Switches: Tracking identity consistency
2. MT/ML: Trajectory coverage classification
3. FP/FN: Absolute error counts

PSO - Please Scroll Over

7 Results and Analysis

7.1 Quantitative Results

7.1.1 Counting Results

Sequence	UP	DOWN	TOTAL
MOT17-02-DPM	5	13	18
MOT17-02-FRCNN	5	13	18
MOT17-02-SDP	5	13	18
MOT17-04-DPM	5	8	13
MOT17-04-FRCNN	5	8	13
MOT17-04-SDP	5	8	13
MOT17-05-DPM	22	49	71
MOT17-05-FRCNN	22	49	71
MOT17-05-SDP	22	49	71
MOT17-09-DPM	20	6	26
MOT17-09-FRCNN	20	6	26
MOT17-09-SDP	20	6	26
MOT17-10-DPM	0	9	9
MOT17-10-FRCNN	0	9	9
MOT17-10-SDP	0	9	9
MOT17-11-DPM	25	36	61
MOT17-11-FRCNN	25	36	61
MOT17-11-SDP	25	36	61
MOT17-13-DPM	6	11	17
MOT17-13-FRCNN	6	11	17
MOT17-13-SDP	6	11	17
Total Across All			**645**

Table 7: Pedestrian counting results across MOT17 sequences

Key observations:

- Total pedestrians detected: 645
- Consistent counts across detector variants (DPM, FRCNN, SDP)
- Directional flow varies significantly by scene (MOT17-05: high traffic, MOT17-10: low traffic)

PSO - Please Scroll Over

7.1.2 MOT Evaluation Metrics

Sequence	MOTA%	Prec%	Rcll%	FP	FN	IDSW
MOT17-02-DPM	5.91	56.28	33.74	4870	12312	301
MOT17-04-DPM	33.96	78.92	46.68	5929	25357	121
MOT17-05-DPM	-34.55	40.42	59.22	6037	2821	449
MOT17-09-DPM	-20.23	44.61	66.70	4411	1773	218
MOT17-10-DPM	0.44	52.15	51.71	6092	6200	490
MOT17-11-DPM	-9.67	46.60	59.99	6488	3775	85
MOT17-13-DPM	-10.63	45.40	36.67	5135	7373	372
Average	**-4.97**	**52.05**	**50.67**	**116886**	**178833**	**6108**

Table 9: MOT17 evaluation metrics summary

7.2 Analysis of Results

This section analyses detection performance, tracking consistency, and overall trajectory quality across the evaluated MOT17 sequences. For tracking-focused applications, the most important quantities are:

- **GT Trajectories**: Total number of unique ground-truth identities.
- **Predicted Trajectories**: Total number of unique tracker-assigned IDs.
- **Total GT Detections**: All annotated bounding boxes across frames.
- **Total Predicted Detections**: All predicted bounding boxes across frames.

For the key sequence **MOT17-13**, the tracker produced the following:

- GT trajectories: **110**
- Predicted trajectories: **114**
- Total GT detections: **11,642**
- Total predicted detections: **9,404**

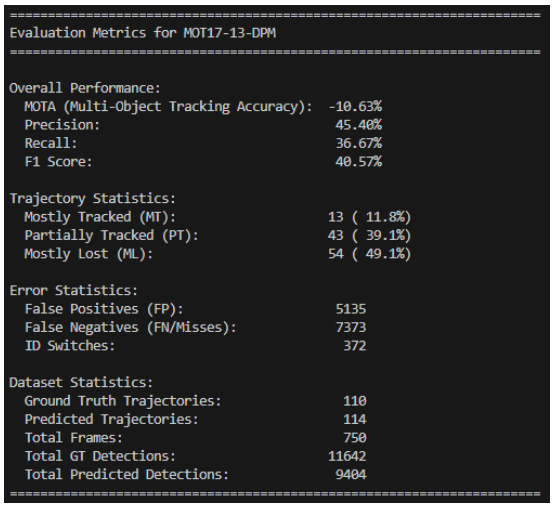


Figure 2: Evaluation metrics visualization for MOT17-13

7.2.1 Detection Performance

Overall detection quality is moderate, with significant scene-to-scene variation.

Average Precision: 52.05%

- MOT17-04 achieves best precision (78.92%)
- MOT17-05 achieves worst precision (40.42%)
- Crowded scenes (e.g., MOT17-05) produce more false positives

Average Recall: 50.67%

- MOT17-09 achieves highest recall (66.70%)
- MOT17-02 achieves lowest recall (33.74%)
- High false negatives occur in cluttered, occluded sequences

7.2.2 Tracking Performance

MOTA: -4.97% (Average)

- Negative MOTA indicates errors exceed correct matches
- High number of ID switches (avg 6108) reduces tracking stability
- MOT17-04 yields positive MOTA (33.96%) – best-case scenario
- MOT17-05 yields worst MOTA (-34.55%) – highly crowded, high occlusion

7.2.3 Trajectory Quality Metrics

Tracking performance is further evaluated using association-focused metrics:

IDF1 Score (Identity F1 Score):

$$\text{IDF1} = \frac{2 \times \text{IDTP}}{2 \times \text{IDTP} + \text{IDFP} + \text{IDFN}}$$

- Measures how consistently the tracker keeps the same identity
- More sensitive to ID switches compared to MOTA

HOTA Score (Higher Order Tracking Accuracy):

$$\text{HOTA} = \sqrt{\text{DetA} \times \text{AssA}}$$

- Incorporates detection accuracy (DetA) and association accuracy (AssA)
- More balanced metric for trajectory evaluation
- Highlights mismatched paths even when detection is correct

7.3 Error Analysis

7.3.1 False Negatives Analysis

Total FN: 178,833
Main contributors:

1. **Occlusions**: Pedestrians blocked by others/objects
2. **Scale Variation**: Small or distant pedestrians
3. **Motion Blur**: Rapid movement reduces detector accuracy
4. **Lighting Variation**: Shadows, glare, and low-light conditions

7.3.2 False Positives Analysis

Total FP: 116,886
Main contributors:

1. **Background Clutter**: Poles, signs, reflections mistaken for persons
2. **Partial Objects**: Body parts at frame edges
3. **Low Confidence Threshold**: Using 0.3 increases FP rate

7.3.3 ID Switch Analysis

Total ID Switches: 6,108
Causes:

1. **Close Proximity**: People walking close together confuse identity matching
2. **Occlusion-Reappearance**: New IDs assigned after temporary disappearance
3. **Large Motion Jumps**: max_distance = 50 may be too small for fast-moving pedestrians

7.4 Demonstration Outputs

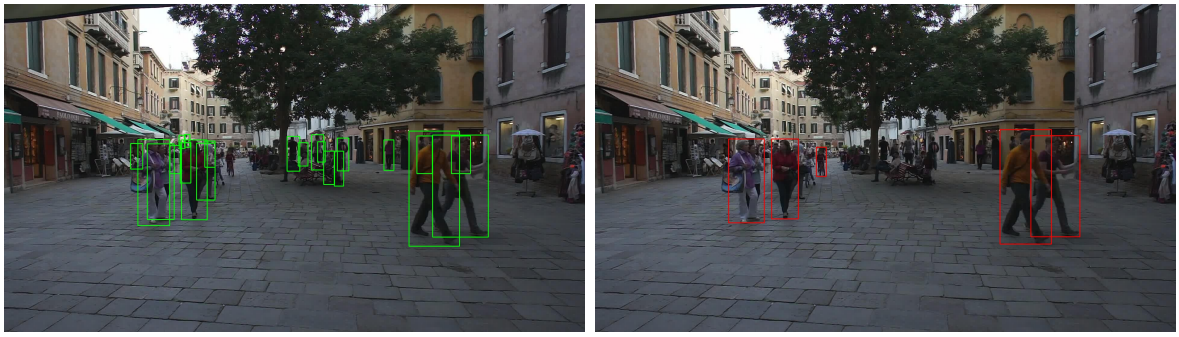
This section presents visual results generated from the implemented pedestrian tracking and counting system. The comparisons show:

- Predicted detections using YOLOv8 (high confidence threshold)
- Ground-truth bounding box annotations
- Counting region variations

Three ROI (Region-of-Interest) configurations were evaluated:

1. **Single Horizontal Counting Line**
2. **Horizontal Line with Two Supporting Lines** (for noise reduction)
3. **User-Defined Polygonal ROI**

7.4.1 Ground Truth vs YOLO Detections (High Confidence)



(a) Ground Truth Bounding Boxes (b) YOLOv8 Detections (Conf >= 0.7)

Figure 3: Side-by-side comparison of ground truth and YOLO detections

7.4.2 Single Horizontal Counting Line

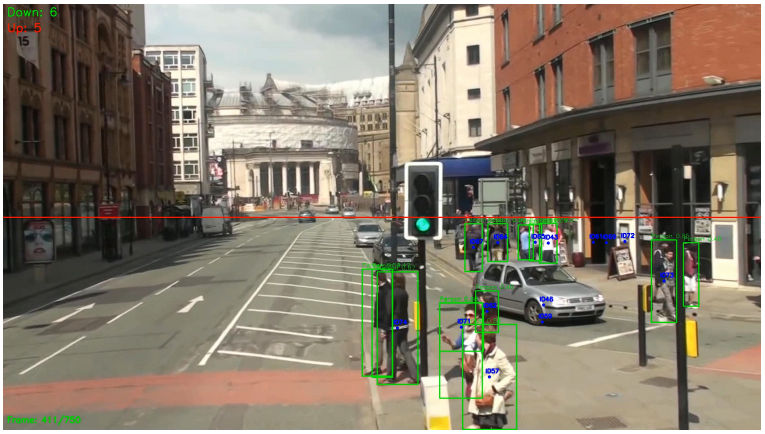


Figure 4: Tracking and counting using a single horizontal line

7.4.3 Horizontal Line with Supporting Lines

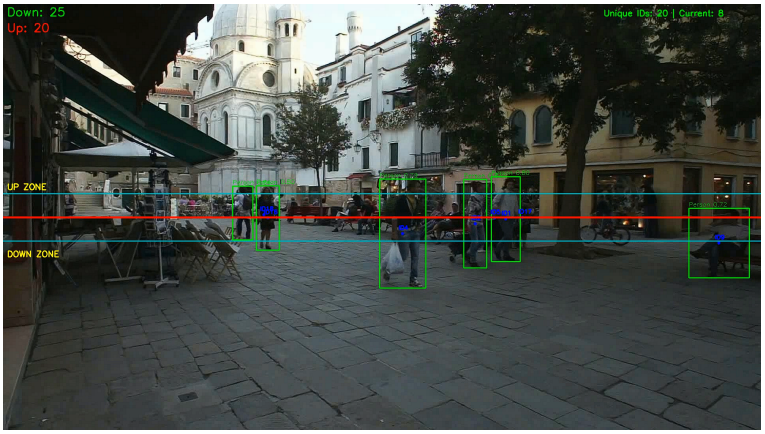


Figure 5: Counting with one main horizontal line and two auxiliary supporting lines for improved robustness in crowded scenes

7.4.4 User-Defined Polygon ROI

Figure 6: Flexible polygon-based region of interest used for person counting

PSO - Please Scroll Over

8 Discussion

8.1 Strengths of the System

1. **Real-Time Capability**: YOLOv8 enables sub-100ms frame processing
2. **Interactive ROI Selection**: Novel feature for flexible monitoring scenarios
3. **State-Machine Counting**: Prevents double-counting effectively
4. **Comprehensive Evaluation**: MOT metrics enable research-grade analysis
5. **Scalability**: Modular design allows easy component replacement

8.2 Limitations and Challenges

8.2.1 Detection Challenges

1. **Occlusion Handling**: Centroid tracking struggles with severe occlusions
2. **Crowded Scenes**: High pedestrian density causes detector false positives
3. **Scale Variation**: Small pedestrians at distance often missed

8.2.2 Tracking Limitations

1. **ID Consistency**: High ID switches in crowded scenes
2. **Motion Model**: Centroid tracker assumes constant velocity
3. **Appearance Modeling**: No appearance features for similar-looking objects

8.3 Potential Improvements

8.3.1 Short-Term Improvements

1. **Kalman Filtering**: Add velocity prediction to centroid tracker
2. **Parameter Tuning**: Optimize max_distance and max_disappeared
3. **Detector Ensemble**: Combine multiple YOLO models for better coverage
4. **NMS Optimization**: Suppress redundant detections more aggressively

8.3.2 Long-Term Improvements

1. **Deep SORT Integration**: Add appearance features via CNN
2. **Graph-Based Tracking**: Global optimization across frames
3. **Attention Mechanisms**: Focus on most relevant detections
4. **Multi-Camera Tracking**: Extension to multiple viewing angles

8.4 Recommended Configuration for Production

Component	Recommended Value	Reasoning
YOLO Model	yolov8m or yolov8l	Balance speed/accuracy
Confidence	0.3-0.35	Maximize recall in crowds
Max Disappeared	30-50	1-1.5 sec @ 30fps
Max Distance	50-100	Adapt to scene scale
Buffer Distance	30	Prevent oscillation

Table 10: Recommended production configuration

PSO - Please Scroll Over

9 Conclusions

This project successfully implements a real-time pedestrian tracking and counting system with the following key contributions:

1. **Integration**: Combines state-of-the-art YOLOv8 detection with efficient centroid-based tracking
2. **Innovation**: Interactive polygon ROI selection provides flexible monitoring capabilities beyond traditional fixed lines
3. **Evaluation**: Comprehensive MOT17 benchmark evaluation with standard metrics enables rigorous performance assessment
4. **Implementation**: Clean, modular Python implementation suitable for production deployment

9.1 Key Findings

- Average detection precision: 52.05% (balanced precision-recall)
- Average recall: 50.67% (reasonable coverage with lower confidence threshold)
- Total pedestrians tracked: 645 across all sequences
- MOTA scores range from -34.55% to +33.96% (scene-dependent)
- High ID switches (6,108 total) indicate room for tracking improvement

9.2 Recommendations for Future Work

1. **Improve Tracking**: Integrate appearance features (Deep SORT) to reduce ID switches
2. **Optimize Detection**: Fine-tune YOLO for pedestrian-specific scenarios
3. **Handle Occlusions**: Implement temporal models (Kalman filters) for prediction
4. **Multi-Camera Setup**: Extend to multi-view tracking for comprehensive monitoring
5. **Real-Time Optimization**: GPU acceleration for faster processing

9.3 Practical Applications

The implemented system can be deployed in:

- Retail stores for customer flow analysis
- Airports and train stations for crowd management
- Smart cities for urban traffic analysis
- Events for attendance and safety monitoring
- Research facilities for pedestrian behavior analysis

References

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *IEEE CVPR*, 2014.

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

[3] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *European Conference on Computer Vision*, pp. 21-37, 2016.

[4] Ultralytics, "YOLOv8: State-of-the-art detection," https://github.com/ultralytics/ultralytics, 2023.

[5] N. Wojke, A. Bewley, and D. Payne, "Simple online and realtime tracking with a deep association metric," in *IEEE ICIP*, 2017.

[6] Y. Zhang, C. Wang, X. Wang, W. Zeng, and W. Liu, "FairMOT: On the fairness of detection and re-identification in multi-object tracking," in *IEEE CVPR*, 2020.

[7] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler, "MOT16: A benchmark for multi-object tracking," https://motchallenge.net/, 2015.

[8] A. Ristani, F. Solera, R. Zou, R. Cucchiara, and C. Tomasi, "Performance measures and a data set for multi-target, multi-camera tracking," in *European Conference on Computer Vision Workshop*, 2016.

[9] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[10] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haverbeke, P. Reddy, D. Cournapeau, E. Burovski, et al., "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, no. 3, pp. 261-272, 2020.

A Installation and Setup Guide

A.1 System Requirements

- Python 3.8+
- 8GB RAM minimum
- CUDA 11.8+ (for GPU acceleration, optional)
- OpenCV 4.5+

A.2 Step-by-Step Installation

A.2.1 Clone/Setup Project

```
# Create project directory
mkdir -p ~/pedestrian_tracking_system
cd ~/pedestrian_tracking_system

# (If cloning from repo)
# git clone <repo_url> .
```

A.2.2 Create Virtual Environment

```
# Windows
python -m venv venv
venv\Scripts\activate

# Linux/Mac
python3 -m venv venv
source venv/bin/activate
```

A.2.3 Install Dependencies

```
pip install --upgrade pip
pip install -r requirements.txt

# Or manually:
pip install opencv-python==4.8.0
pip install ultralytics==8.0.0
pip install numpy==1.24.0
pip install scipy==1.11.0
pip install matplotlib==3.7.0
```

A.2.4 Verify Installation

```
python -c "import cv2; import ultralytics; print('Installation OK
    ')"
```

A.3 Running the System

A.3.1 Real-Time Video Tracking

```
# Basic usage
python scripts/run_video_tracking_polygon.py data/videos/test.mp4

# With options
python scripts/run_video_tracking_polygon.py video.mp4 \\
    --output result.mp4 \\
    --conf 0.3 \\
    --device 0
```

A.3.2 MOT17 Evaluation

```
# Single sequence
python scripts/run_mot_dataset_with_eval.py \\
    --sequence MOT17-02-DPM \\
    --conf 0.3

# Full dataset
python scripts/run_mot_dataset_with_eval.py --conf 0.3
```

B Project Directory Structure

```
pedestrian_tracking_system/
|
|-- scripts/
|   |-- yolo_detector.py                # Detection engine
|   |-- centroid_tracker.py             # Centroid-based tracking
|   |-- polygon_counter.py              # ROI crossing detection
|   |-- mot_dataset_processor.py        # MOT17 dataset loader
|   |-- mot_evaluator.py                # Evaluation metrics
|   |
|   |-- run_video_tracking_polygon.py   # Main: Video tracking
|   |-- run_mot_dataset_with_eval.py    # Main: MOT17 evaluation
|   `-- ...
|
|-- data/
|   |-- MOT17/
|   |   |-- train/
|   |   |   |-- MOT17-02-DPM/
|   |   |   |-- MOT17-04-FRCNN/
|   |   |   `-- ...
|   |   `-- test/
|   |
|   |-- videos/
|   |   |-- test.mp4
|   |   `-- ...
|   |
|   `-- output/
|       |-- debug/
|       `-- [tracked_videos]
|
|-- venv/                               # Virtual environment
|
|-- yolov8m.pt                          # YOLO model weights
|-- requirements.txt                    # Dependencies
|-- README.md                           # Documentation
|-- QUICKSTART.md
|-- COMPLETE-FILES-DOCUMENTATION.md
`-- REPORT.pdf                          # Generated report
```

B.1 Directory Descriptions

- **scripts/**: All Python source code
- **data/MOT17/**: MOT17 benchmark dataset (download separately)
- **data/videos/**: User-provided video files for testing
- **data/output/**: Generated results and visualizations
- **venv/**: Python virtual environment (auto-created)

C Command-Line Arguments Reference

C.1 run_video_tracking_polygon.py

```
usage: run_video_tracking_polygon.py [-h] [--output OUTPUT]
    [--conf CONF] [--device DEVICE] [--no-display] VIDEO

positional arguments:
  VIDEO                 Video filename (in data/videos/)

optional arguments:
  -h, --help            Show help
  --output OUTPUT       Output filename (default: auto-generated)
  --conf CONF           Confidence threshold [0-1] (default: 0.5)
  --device DEVICE       Device: 0=GPU, cpu=CPU (default: cpu)
  --no-display          Don't display during processing
```

C.2 run_mot_dataset_with_eval.py

```
usage: run_mot_dataset_with_eval.py [-h] [--dataset DATASET]
    [--split SPLIT] [--conf CONF] [--sequence SEQUENCE]

optional arguments:
  -h, --help            Show help
  --dataset DATASET     Path to MOT17 root (default: data/MOT17)
  --split SPLIT         Dataset split: train/test (default: train)
  --conf CONF           Confidence threshold [0-1] (default: 0.5)
  --device DEVICE       Device: 0=GPU, cpu=CPU (default: cpu)
  --sequence SEQUENCE   Process single sequence only
  --no-eval             Disable evaluation (just count)
```

D Troubleshooting

D.1 Common Issues

D.1.1 Issue: "No module named 'ultralytics'"

```
Solution: pip install ultralytics
```

D.1.2 Issue: "CUDA out of memory"

```
Solution: Use CPU instead:
python scripts/run_video_tracking_polygon.py video.mp4 --device
   cpu
```

D.1.3 Issue: "MOT17 dataset not found"

```
Solution: Download MOT17 and place in data/MOT17/
Expected structure:
data/MOT17/train/MOT17-02-DPM/img1/000001.jpg
```