Beep: Left Behind Items Detection in Autonomous Public Transport

Sahil Bishnoi

gtid: 903942137

I. Introduction

The purpose of this project is to develop a computer vision system that can detect, track, and identify personal items left behind in autonomous public transport vehicles. This system is designed to enhance security and convenience for passengers by alerting them and the transport vehicle authorities when an item is left behind. The project leverages the YOLOv7 object detection model and uses entry-line and proximity-based methods to pair items with their owners. The code for this project is available at the **Left Behind Items GitHub Repository**

II. Objective

The primary objective of this project is to develop a reliable computer vision system that:

1. Detects and tracks passengers and their belongings.
2. Pairs detected items with their respective owners.
3. Alerts when an item is left behind after the owner exits the vehicle.

III. Methodology

A. Data Collection

Video footage from two cameras (inside and rear of the vehicle) was used for this project. The cameras recorded multiple sessions, providing varied scenarios for testing the system.

B. Model Selection

The YOLOv7 model trained on the COCO dataset was selected for its accuracy and efficiency in object detection tasks. The Bytetrack algorithm was used to maintain consistent tracking of detected objects across frames.

What is YOLOv7?

The YOLO (You Only Look Once) v7 model is part of the YOLO family of single-stage object detectors. In a YOLO model, image frames are featurized through a backbone. These features are combined and mixed in the neck, and then passed along to the head of the network, where YOLO predicts the locations and classes of objects for which bounding boxes should be drawn. YOLO performs post-processing via non-maximum suppression (NMS) to arrive at its final predictions.

YOLO models are widely used by the computer vision and machine learning communities for object detection due to their small, nimble architecture and the ability to train on a single GPU. The compact architecture and real-time inference speed allow practitioners to allocate minimal hardware resources to power their applications.

What is the COCO Dataset?

The COCO (Common Objects in Context) dataset is a large-scale image recognition dataset used for object detection, segmentation, and captioning tasks. It contains over 330,000 images, each annotated with 80 object categories and five captions describing the scene. The COCO dataset is widely used in computer vision research and has been instrumental in training and evaluating many state-of-the-art object detection and segmentation models.

The dataset has two main components: the images and their annotations.

- Images: Organized into a hierarchy of directories, with the top-level directory containing subdirectories for the train, validation, and test sets.
- Annotations: Provided in JSON format, with each file corresponding to a single image. Each annotation includes the following information:
    - Image file name
    - Image size (width and height)
    - List of objects with the following details:
        - Object class (e.g., "person," "car")
        - Bounding box coordinates (x, y, width, height)
        - Segmentation mask (polygon or RLE format)
        - Keypoints and their positions (if available)
    - Five captions describing the scene

What is ByteTrack?

- ByteTrack is a Multi-Object Tracking (MOT) framework that associates every detection box, including low-confidence ones, to improve tracking consistency.
- Keeps non-background low-confidence detection boxes for secondary association, enhancing tracking even with occlusions or appearance changes.
- Works with any object detection method (e.g., YOLO, RCNN) and instance association methods (e.g., IoU, feature similarity).

C. Video Processing Pipeline

Two Python scripts (left_behind_inside_camera.py and left_behind_rear_camera.py) were developed to process the video files.

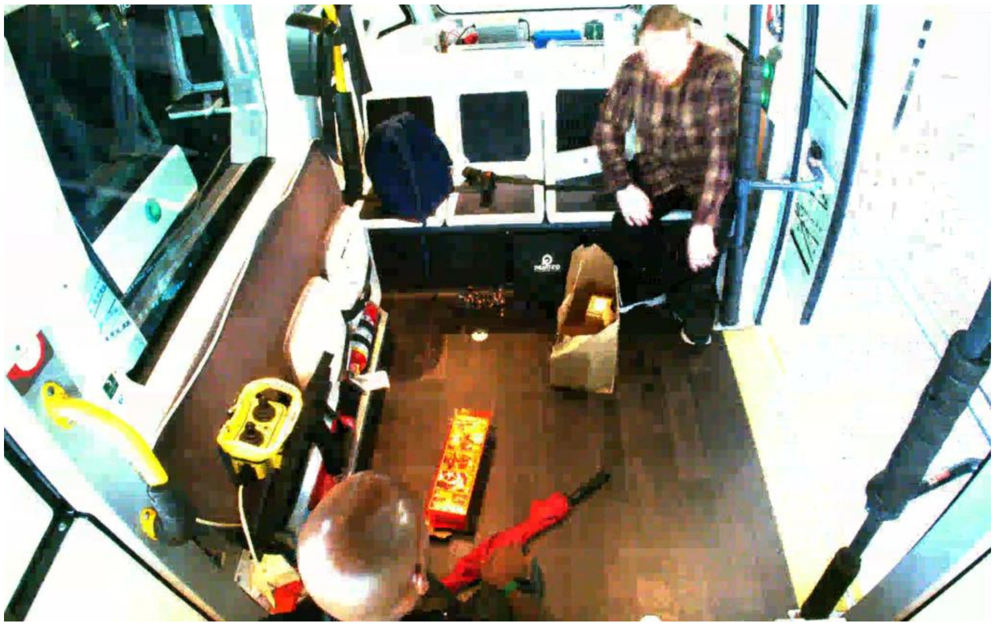

Image 1: Raw video frame from the inside camera

Each script performs the following tasks:

1. **Object Detection**: Using YOLOv7 to detect persons and objects in each frame.
2. **Tracking**: Employing the ByteTrack algorithm to maintain consistent tracking of detected objects across frames.

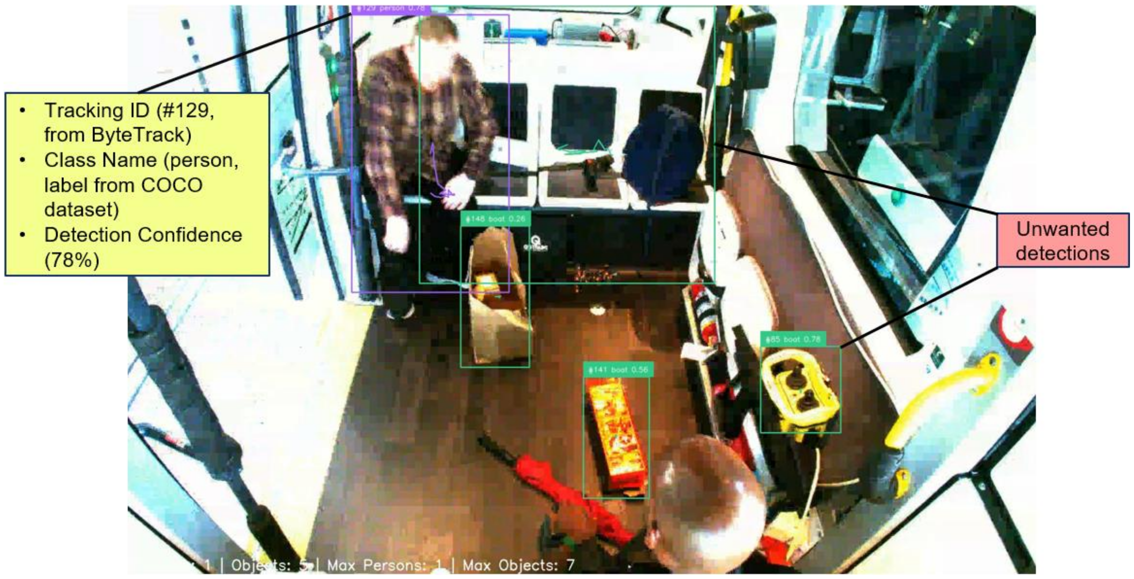
#129 person 0.78
Tracking ID (#129, from ByteTrack)
Class Name (person, label from COCO dataset)
Detection Confidence (78%)
Unwanted detections
#148 boat 0.26
#141 boat 0.56
#85 boat 0.78
1 | Objects: 5 | Max Persons: 1 | Max Objects: 7

Image 2: Annotated frame with detections before filtering

3. **Noise Reduction**: The real-time video feed from the vehicle is often not of the highest quality. To address this, several noise reduction techniques were implemented:

   - Class-Based Filtering: Irrelevant detections are filtered out based on class IDs.

```python
# Filter detections based on class IDs
tv_class_id = results.names.index("tv")
detections = detections[detections.class_id != tv_class_id]
```

   - Size-Based Filtering: Unwanted large detections are filtered out based on the relative size of the bounding box compared to the frame size.

```python
# Filter detections based on relative area
image_area = frame.shape[0] * frame.shape[1]
detections = detections[(detections.area / image_area) < 0.1]
```

   - Polygon Zone Filtering: A polygon zone is created to filter out all detections outside the specified zone, counting only objects and people within the zone. This helps reduce noise from irrelevant objects.

```python
# Define the polygon zone with coordinates.
polygon = np.array([[0, 1200], [400, 150], [1300, 150], [1450, 1200]])


# Initialize the polygon zone with frame dimensions.
    if polygon_zone is None:
        frame_height, frame_width = frame.shape[:2]
        polygon_zone = sv.PolygonZone(polygon=polygon,
                       frame_resolution_wh=(frame_width, frame_height))
        polygon_zone_annotator = sv.PolygonZoneAnnotator(zone=polygon_zone,
                                 color=sv.Color.YELLOW, thickness=2)


# Apply the polygon zone mask to the detections.
    polygon_mask = polygon_zone.trigger(detections=detections)
    detections_in_zone = sv.Detections(
        xyxy=detections.xyxy[polygon_mask],
        confidence=detections.confidence[polygon_mask],
        class_id=detections.class_id[polygon_mask],
        tracker_id=detections.tracker_id[polygon_mask]
    )
```

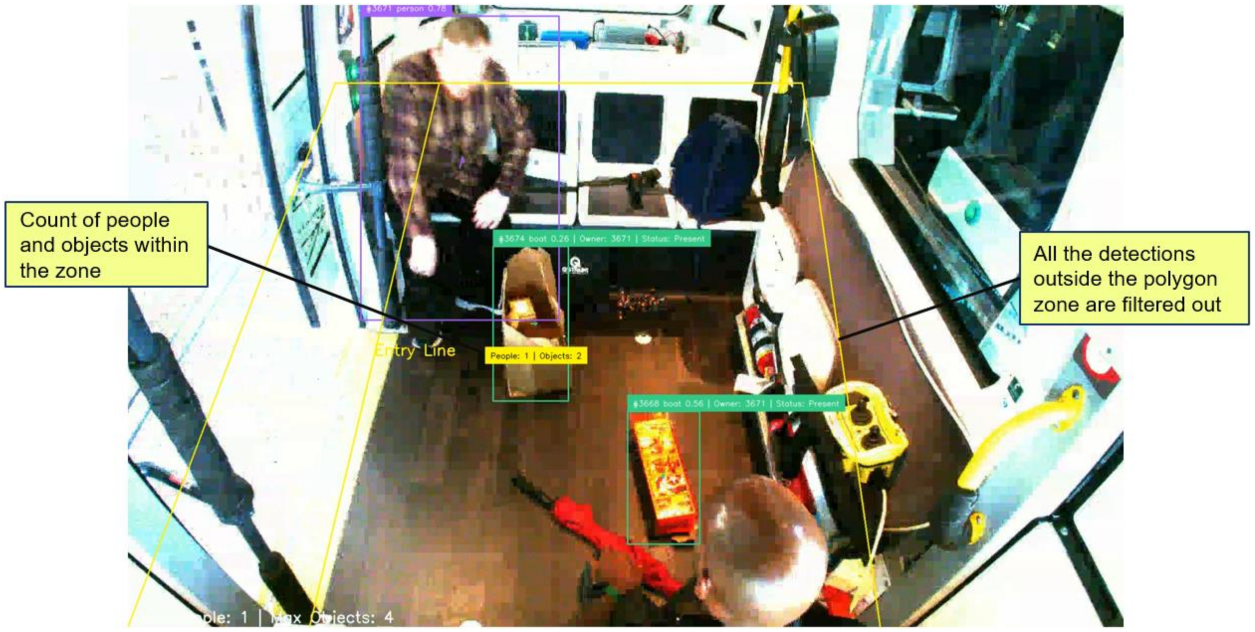
Count of people and objects within the zone
All the detections outside the polygon zone are filtered out
Entry Line
People: 1 | Objects: 2

Image 3: Noise reduction by filtering unwanted detections

4. **Item-Person Association:** We want to correctly pair the item with its owner, and for that, we are using two methods: entry line ownership assignment and proximity-based assignment. Below is an explanation of each method:

   Entry Line Based Assignment

   The entry line ownership assignment is used to pair an item with its owner when both cross the entry line simultaneously or in quick succession.

   - The entry line is defined by two points in the frame:

     ```
     # Define the entry line coordinates.
     entry_line = [(350, 1200), (600, 150)]
     ```

   - The crosses_entry_line function checks if a bounding box (person or object) crosses the predefined entry line. This is done by checking if any part of the bounding box intersects the entry line.

```python
# Function to check if a bounding box crosses the entry line.
def crosses_entry_line(bbox, line):
    """
    Checks if a bounding box crosses the entry line.

    Args:
    bbox (list): Bounding box coordinates [x1, y1, x2, y2].
    line (list): Entry line coordinates [(x1, y1), (x2, y2)].

    Returns:
    bool: True if the bounding box crosses the entry line, False otherwise.
    """
    x1, y1, x2, y2 = bbox
    line_x1, line_y1, line_x2, line_y2 = line[0][0], line[0][1], line[1][0],
                                          line[1][1]
    return ((x1 <= line_x1 <= x2 or x1 <= line_x2 <= x2) and (y1 <= line_y1 <= y2
            or y1 <= line_y2 <= y2))
```

- During each frame, the code checks if any person or object crosses the entry line. If a bounding box crosses the line, it is recorded in the entry_line_crossed dictionary.

```python
# Assign ownership based on entry line crossing.
    for obj_id, obj_crossed in entry_line_crossed.items():
        if obj_crossed == 'object':
            for person_id, person_crossed in entry_line_crossed.items():
                if person_crossed == 'person':
                    detected_entries.append((person_id, obj_id))
                    break

    # Assign ownership if both person and object are detected crossing the entry
line.
    for person_id, obj_id in detected_entries:
        if person_id in person_detections.tracker_id and obj_id in
                                                   object_detections.tracker_id:
            object_to_person[obj_id] = person_id
            owner_assigned_time[obj_id] = time.time()
```

- If both a person and an object are detected crossing the entry line within a certain time frame, their IDs are paired and stored in detected_entries. Once an object is assigned to a person using the entry line method, the assignment is permanent for the duration of the tracking session. This ensures that their association remains intact even if they move around.

```
# Assign ownership based on entry line crossing.
    for obj_id, obj_crossed in entry_line_crossed.items():
        if obj_crossed == 'object':
            for person_id, person_crossed in entry_line_crossed.items():
                if person_crossed == 'person':
                    detected_entries.append((person_id, obj_id))
                    break

    # Assign ownership if both person and object are detected crossing the entry 
line.
    for person_id, obj_id in detected_entries:
        if person_id in person_detections.tracker_id and obj_id in 
                                                                object_detections.tracker_id:
            object_to_person[obj_id] = person_id
            owner_assigned_time[obj_id] = time.time()
```

Proximity-Based Assignment

The proximity-based assignment function is used when the entry line logic does not capture the detection of both a person and their associated object simultaneously. This function ensures that objects are assigned to the nearest detected person based on the distance between their bounding boxes.

```
def assign_possession(person_detections, object_detections, object_to_person):
    """
    Assigns objects to the nearest detected person permanently.

    Args:
    person_detections (sv.Detections): Detections of persons.
    object_detections (sv.Detections): Detections of objects.
    object_to_person (dict): Dictionary mapping object IDs to person IDs.

    Returns:
    dict: Updated dictionary mapping object IDs to person IDs.
    """
    for i, obj in enumerate(object_detections.xyxy):
        obj_id = object_detections.tracker_id[i]
        if obj_id not in object_to_person:
            obj_center = [(obj[0] + obj[2]) / 2, (obj[1] + obj[3]) / 2]
            min_distance = float('inf')
            assigned_person = None
            for j, person in enumerate(person_detections.xyxy):
                person_center = [(person[0] + person[2]) / 2, (person[1] +
                                                               person[3]) / 2]
                distance = np.linalg.norm(np.array(obj_center) -
                                                      np.array(person_center))
                if distance < min_distance:
                    min_distance = distance
                    assigned_person = person_detections.tracker_id[j]
            if assigned_person is not None:
                object_to_person[obj_id] = assigned_person
                owner_assigned_time[obj_id] = time.time()
    return object_to_person
```

- The function iterates over each detected object (object_detections.xyxy). For each object, it retrieves its unique identifier (obj_id).
- The function checks if the object is already assigned to a person. If not, it proceeds with the assignment.
- The center of the object's bounding box is calculated using its coordinates.
- A variable min_distance is initialized to infinity to keep track of the smallest distance found.
- The function iterates over each detected person (person_detections.xyxy). For each person, it calculates the center of their bounding box.
- The Euclidean distance between the center of the object and the center of the person is computed using np.linalg.norm.
- If the calculated distance is smaller than the current min_distance, the function updates min_distance and sets assigned_person to the current person's ID.
- After finding the nearest person, the function assigns the object to this person and records the assignment time.
- Similar to the entry line method, once an object is assigned to a person using the proximity method, the assignment is permanent for the duration of the tracking session. This ensures that their association remains intact even if they move around. Alert Logic

Example Scenario: Entry Line Missed: A person crosses the entry line with an object, but the object is not detected. Later, another person crosses, and only their object is detected. Proximity Assignment: The system will use the proximity-based assignment to link objects to the nearest detected person, ensuring ownership is accurately established even when entry line detection fails.

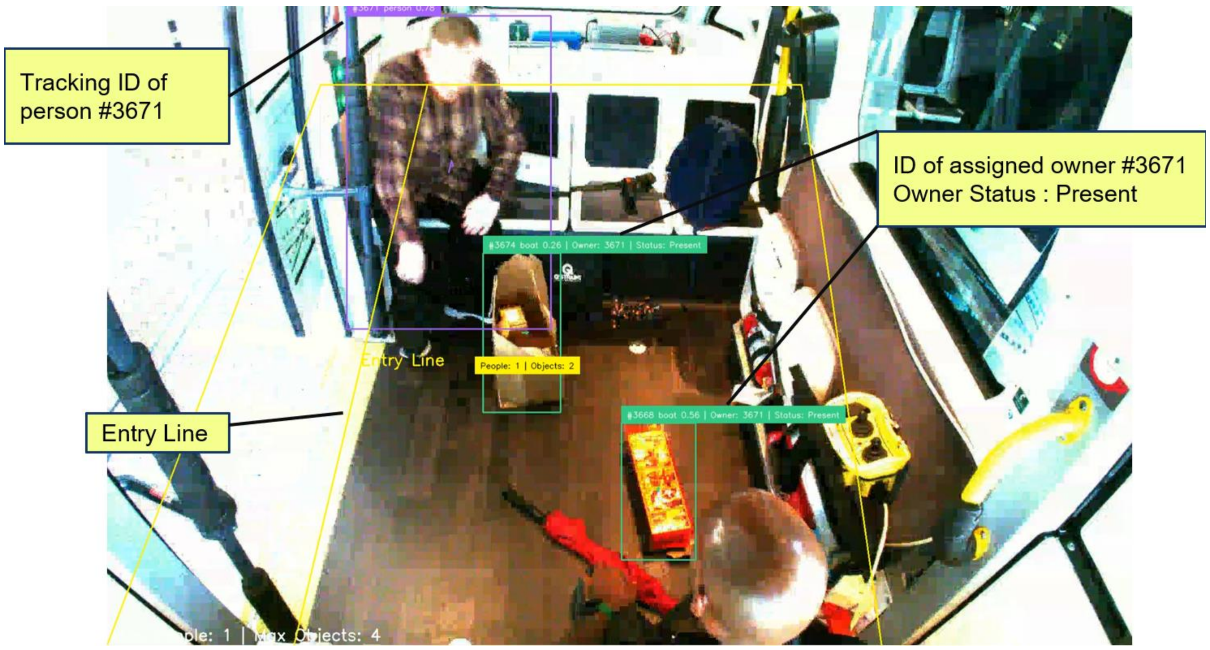
Tracking ID of person #3671
#3671 person 0.78
ID of assigned owner #3671 Owner Status : Present
#3674 boat 0.26 | Owner: 3671 | Status: Present
Entry Line
People: 1 | Objects: 2
#3668 boat 0.56 | Owner: 3671 | Status: Present
Entry Line
ple: 1 | Max Objects: 4

Image 4: Item-Owner Association

5. Alert Logic

The alert logic in the code is designed to trigger an alert if an object is left behind by its assigned owner.

```
# Check if the assigned person for any object is no longer present.
    alert_triggered = False
    persons_present = set(detections_in_zone.tracker_id[detect
                         -ions_in_zone.class_id == person_class_id])
    for obj_id, person_id in object_to_person.items():
        if person_id not in persons_present:
            if obj_id in owner_assigned_time:
                if obj_id not in missing_times:
                    missing_times[obj_id] = current_time
                elif current_time - missing_times[obj_id] >= alert_duration:
                    alert_triggered = True
                    break
        else:
            if obj_id in missing_times:
                del missing_times[obj_id]

# Add alert message if triggered.
    if alert_triggered:
        alert_text = "ALERT: Item left behind!"
        alert_position = (10, 50)
        alert_color = (0, 0, 255)  # Red color for alert
        cv2.putText(annotated_frame, alert_text, alert_position, cv2.FONT_HERS
                                   -HEY_SIMPLEX, font_scale, alert_color, thickness)
```

- 'alert_duration' variable defines the duration (in seconds) after which an alert is triggered if an object is detected without its assigned owner.
- During each frame, the code maintains a list of 'persons_present', which contains the tracker IDs of all persons detected in the current frame within the polygon zone.
- The code iterates through the dictionary 'object_to_person', which maps object IDs to their assigned person IDs. -For each object, it checks if the assigned person is still present in the current frame.
- If the assigned person is not detected in the current frame ('person_id not in persons_present'):
    - The code checks if the object has been missing for a significant amount of time.
    - If the object was not previously recorded as missing, it records the current time in missing_times for that object.
    - If the object was already missing, the code calculates the duration it has been missing by comparing the current time with the recorded missing time.
    - If the object has been missing for longer than alert_duration, an alert is triggered.

- If an object has been left behind (i.e., its assigned person has been missing for longer than 'alert_duration'), the 'alert_triggered flag' is set to True. The alert message "ALERT: Item left behind!" is added to the annotated frame.
- If the assigned person is present in the current frame, any previously recorded missing time for that object is removed from missing_times.

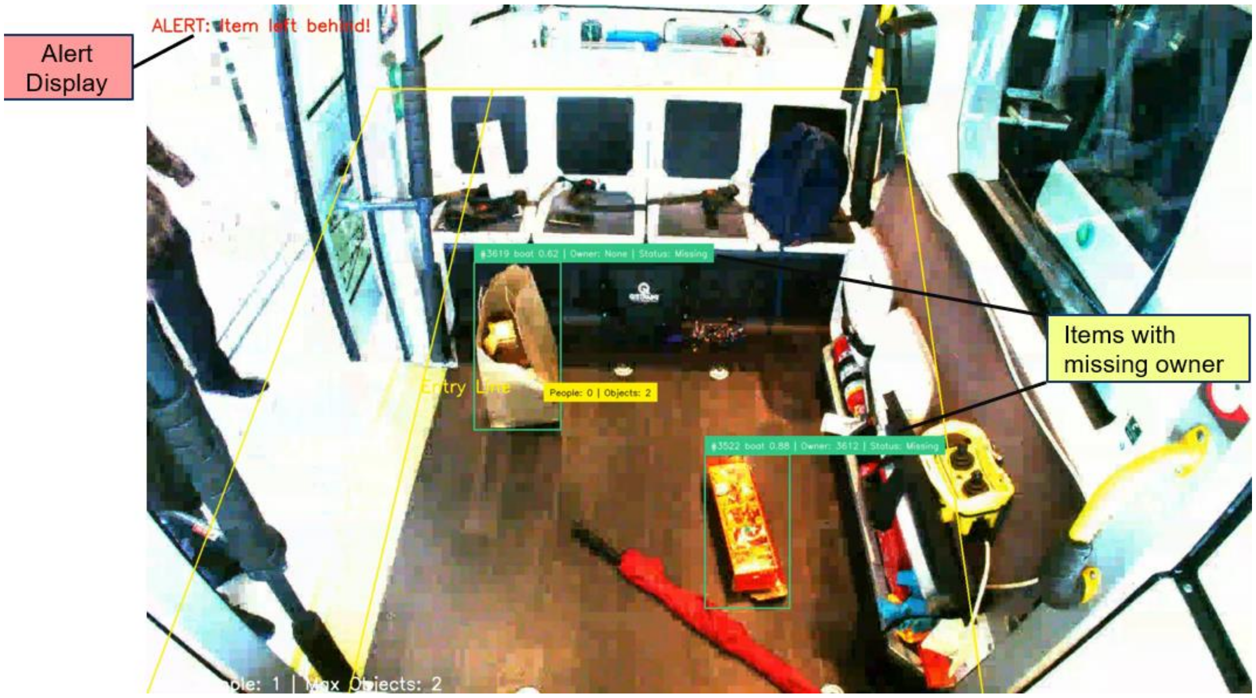

Image 5: Alert pop up in the top left corner when an item is detected without its owner

IV. Results

The system was tested on multiple video files, demonstrating its capability to:

- Accurately detect and track persons and objects.
- Correctly pair items with their owners using entry line and proximity-based methods.
- Effectively trigger alerts when items are left behind.

V. Discussion

The project demonstrates the feasibility of using computer vision for enhancing security in public transport. The combination of entry line and proximity-based methods ensures robust ownership assignment. However, the system relies on continuous detection, and any failure in object detection could impact the alert mechanism.

Recommendations

- **Improve Detection Accuracy**: While YOLOv7 is trained on the popular COCO dataset, it is primarily concerned with detecting general items. To improve detection accuracy, we can train the YOLO model on a custom dataset comprising images of common items from the camera feed. This training will account for factors such as camera quality, lighting conditions, and camera angles.
- **Real-Time Processing**: Ensuring real-time processing of video feeds may require optimization of the detection and tracking algorithms.
- **Integration with Transport Systems**: Integration with existing transport management systems to provide real-time alerts to passengers and authorities.
- **User Feedback**: Incorporating feedback from users can help refine the alert mechanisms and improve overall system performance.

VI. Conclusion

This project successfully developed a system to detect and track items left behind in public transport, enhancing passenger security and convenience. The use of YOLOv7 for object detection, combined with entry line and proximity-based assignment methods, ensures accurate pairing of items with their owners. Future work can build on this foundation to create even more robust and user-friendly systems.

VII. References

- YOLOv7 GitHub Repository
- ByteTrack GitHub Repository
- Supervision GitHub Repository