Object Detection and Tracking using Azure Computer Vision/Custom Vision

Kshitij Buch (225437881)

SIG788 – Engineering AI Solutions

06/04/2025

PART 1 : Object Detection Using Azure Computer Vision API

1. Introduction

Azure Computer Vision is a cloud-based AI service that enables developers to analyze visual content. In this task, the first part focuses exclusively on detecting objects within static images using the Azure Computer Vision REST API. The objective is to extract object types and bounding box coordinates, and visualize them using Python in Jupyter Notebook.

2. Object Detection Using Azure Computer Vision API

The approach taken involves integrating Azure's REST API with Python code to detect objects in user-provided images. This part includes setting up an Azure Computer Vision resource, uploading an image through a Jupyter Notebook widget, and drawing bounding boxes with classification labels.

Steps followed:

1. Created a Computer Vision resource in the East US region to support Vision API.
2. Used Vision Studio to validate image object detection interactively.
3. Developed a Jupyter Notebook using Python libraries: `requests`, `PIL`, `matplotlib`, and `ipywidgets`.
4. Allowed the user to upload a local image file through an interactive file upload widget in the notebook (ipywidgets.FileUpload).
5. Made REST API calls to analyze images and return object details.
6. Parsed JSON response to draw bounding boxes and label detected objects with enhanced font size using `ImageFont.truetype`.
7. Displayed final images with overlaid labels and bounding boxes, and formatted output with clear textual summaries.

3. Preliminary Evaluation and Limitations

The Azure Computer Vision API demonstrated high accuracy in detecting common objects in diverse scenes. Confidence scores were generally above 0.75 for clearly visible objects.

However, the model showed some notable limitations:

- It failed to detect partially occluded or side-facing people (e.g., the woman in Image2).
- Overlapping or small items are sometimes missed or misclassified.
- There is no native way to fine-tune detection results or suppress false positives when using the standard Computer Vision API.
- Unlike Azure Custom Vision, the standard Azure Computer Vision API does not support training or fine-tuning on custom object classes. This makes it suitable only for general-purpose detection of common object types.

Despite these limitations, the API is effective for general-purpose object detection and offers a straightforward integration experience.

3.1 Evaluation of Detection Accuracy

The Azure Computer Vision API showed robust performance in detecting commonly occurring objects like people, dogs, cars, and bicycles in various scenes. However, a deeper evaluation across three test images revealed both strengths and some limitations in detection accuracy and labeling consistency.

Detection Strengths

- In Image1, indoor objects such as a *dog*, *plant*, and *chair* were correctly detected with confidence scores above 0.70.
- In Image2, the system successfully detected two *persons* and a *golden retriever*, with one of the people recognized with a high confidence of 0.91.

- In Image3, Azure detected multiple *cars*, a *bicycle*, a *cyclist*, and a *passenger was detected inside a car which was actually the reflection of the person on the bicycle*:
    - The detection of moving and stationary objects in a busy urban street showed the model's general-purpose capability.
    - High confidence scores were noted for many vehicles (e.g., car at 0.89, *person at 0.90*).

Detection Limitations

- In Image2, the woman on the left was not detected, likely due to her side-facing pose and occlusion by the dog and other person.
- In Image3, although the major objects were captured, some key issues were observed:
    - The middle car, partially occluded by the person on the bicycle, was not identified as a single 'car' but rather as two separate "land vehicle" objects with lower confidence scores (0.51 and 0.52).
    - This suggests the model struggles with occluded or overlapped objects, leading to fragmented detection or duplicated labels.
    - There was inconsistent labeling across similar objects — the same vehicle type was detected as both *car* and *land vehicle* in the same image.
    - The model also detected reflection of person on bicyle on the car's window as a separate object which was false positive.

In some cases, the **object boundaries were clipped**, or **text labels overlapped** in the image, affecting clarity.

4. Observations on Azure Computer Vision

Strengths:
- Easy REST API integration
- High confidence for common object types
- Fast and scalable

Limitations:
- Limited control over the model (no fine-tuning)
- Cannot detect custom object classes
- May miss occluded subjects or return clipped labels and false positives too.

If the task required detection of domain-specific objects (e.g., medical devices, rare animals), Azure Custom Vision would be a better alternative due to its support for training and model refinement which we shall see in use in next part of object tracking.

5. Visual Outputs and Screenshots

Screenshots captured during the execution of this task include the following stages:

- Creation of the Azure Computer Vision resource in the East US region via the Azure Portal
- Selection and confirmation of the resource within Azure Vision Studio
- Jupyter Notebook interface showing the file upload widget and image selection
- Detected objects overlaid on uploaded images, with improved bounding box font size using the PIL.ImageFont module
- Terminal output displaying object classification labels and their confidence scores
- Detected object coordinates (X, Y, Width, Height) were extracted from the API response and printed alongside object labels and confidence scores to understand spatial placement in the image
- Partial JSON response from the Azure REST API showcasing the structure of the detected object data

These screenshots provide visual evidence of the detection process and its effectiveness. All outputs are annotated and used for interpretation and evaluation in the subsequent sections of this report.

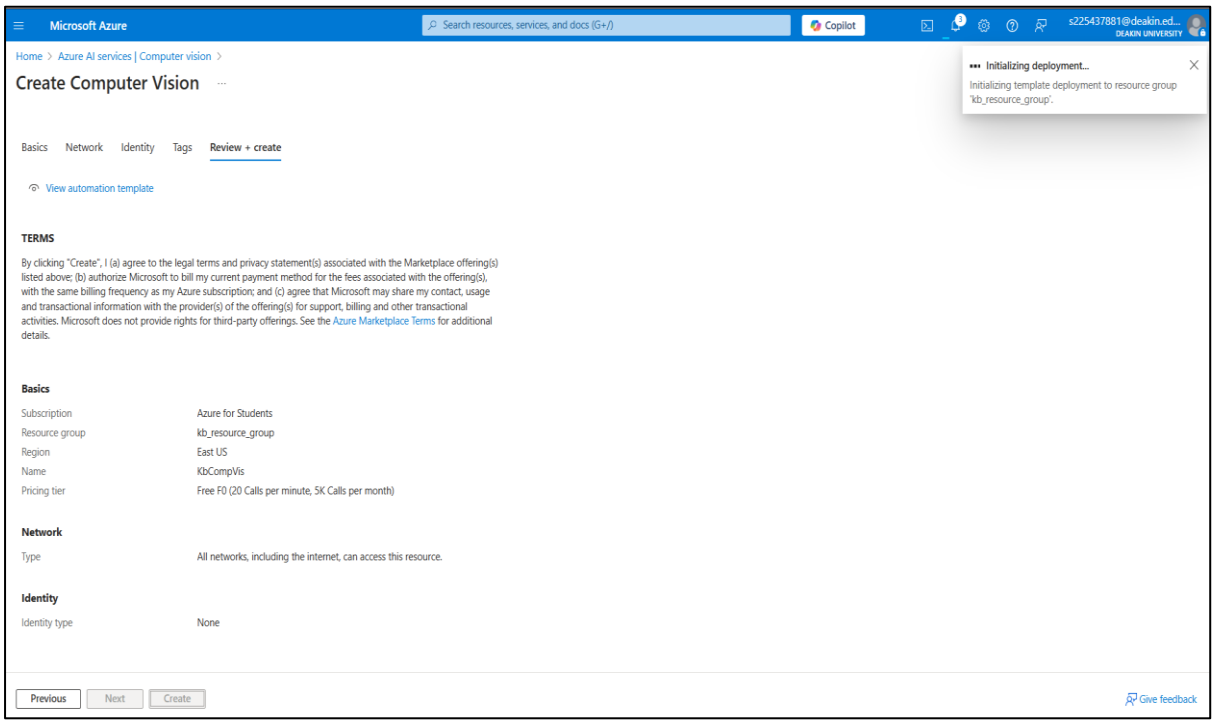


Fig1: Computer Vision Resource Creation

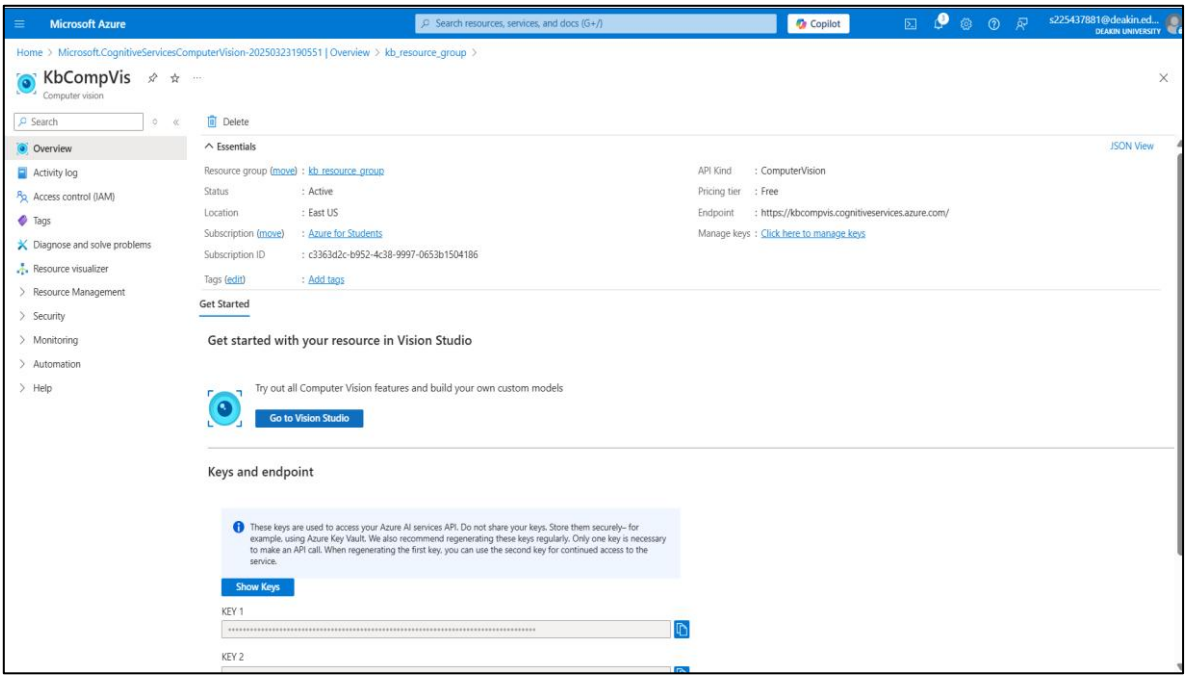


Fig2: To Go To Computer Vision Studio Platform

To access Vision Studio, the one has to select the created Computer Vision resource (KbCompVis) from the Azure portal and click the **"Go to Vision Studio"** button located in the **"Get started with your resource"** section as seen in Fig2.

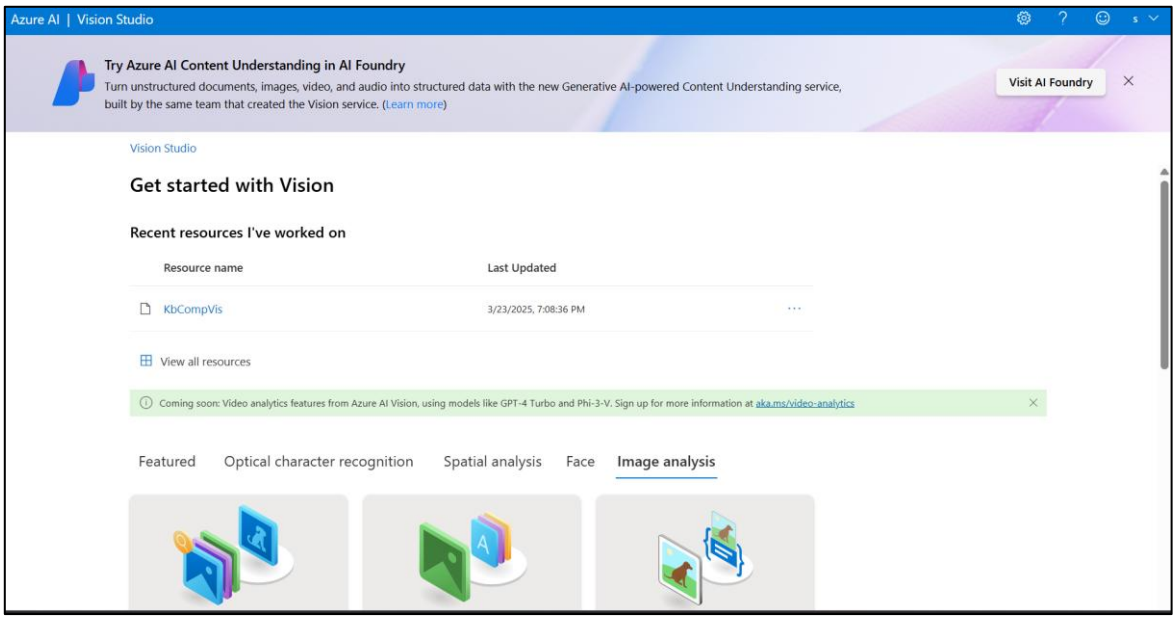
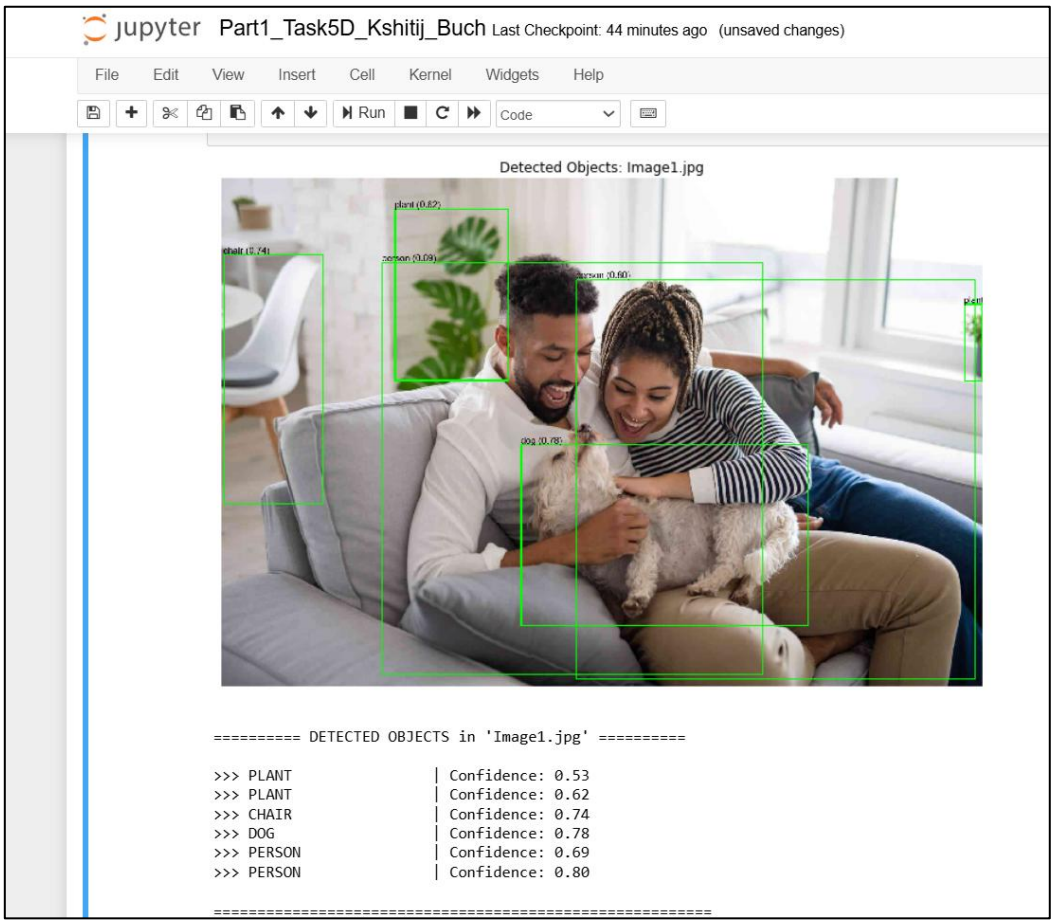


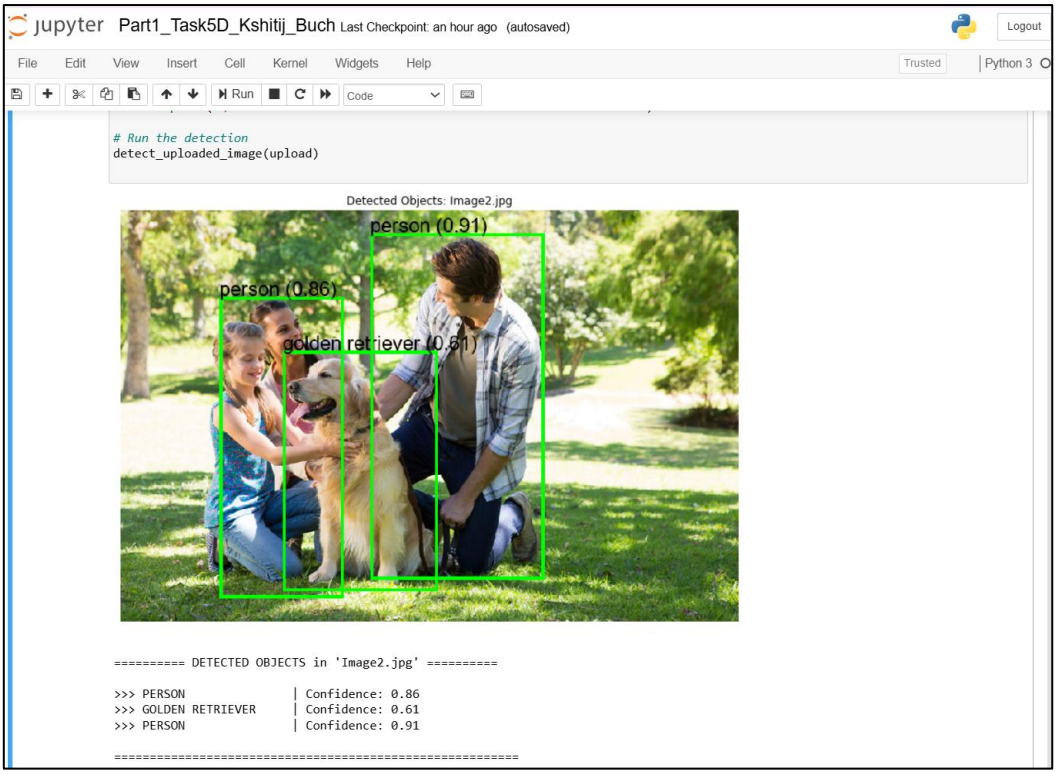
Fig3: Selection and confirmation of the resource within Azure Vision Studio

You can see the kbCompVis resource successfully mapped to Vision Studio as seen above.

Fig4: Image Upload Feature Shown In Jupyter Notebook

It is a good practice to take the image from user which he/she wants to analyse and know more details about. So that upload functionality is seen provided in the Jupyter Notebook.


jupyter Part1_Task5D_Kshitij_Buch Last Checkpoint: 44 minutes ago (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Run Code
Detected Objects: Image1.jpg
========== DETECTED OBJECTS in 'Image1.jpg' ==========
>>> PLANT | Confidence: 0.53
>>> PLANT | Confidence: 0.62
>>> CHAIR | Confidence: 0.74
>>> DOG | Confidence: 0.78
>>> PERSON | Confidence: 0.69
>>> PERSON | Confidence: 0.80


jupyter Part1_Task5D_Kshitij_Buch Last Checkpoint: an hour ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted Python 3
Run Code
# Run the detection
detect_uploaded_image(upload)
Detected Objects: Image2.jpg
person (0.91)
person (0.86)
golden retriever (0.61)
========== DETECTED OBJECTS in 'Image2.jpg' ==========
>>> PERSON | Confidence: 0.86
>>> GOLDEN RETRIEVER | Confidence: 0.61
>>> PERSON | Confidence: 0.91

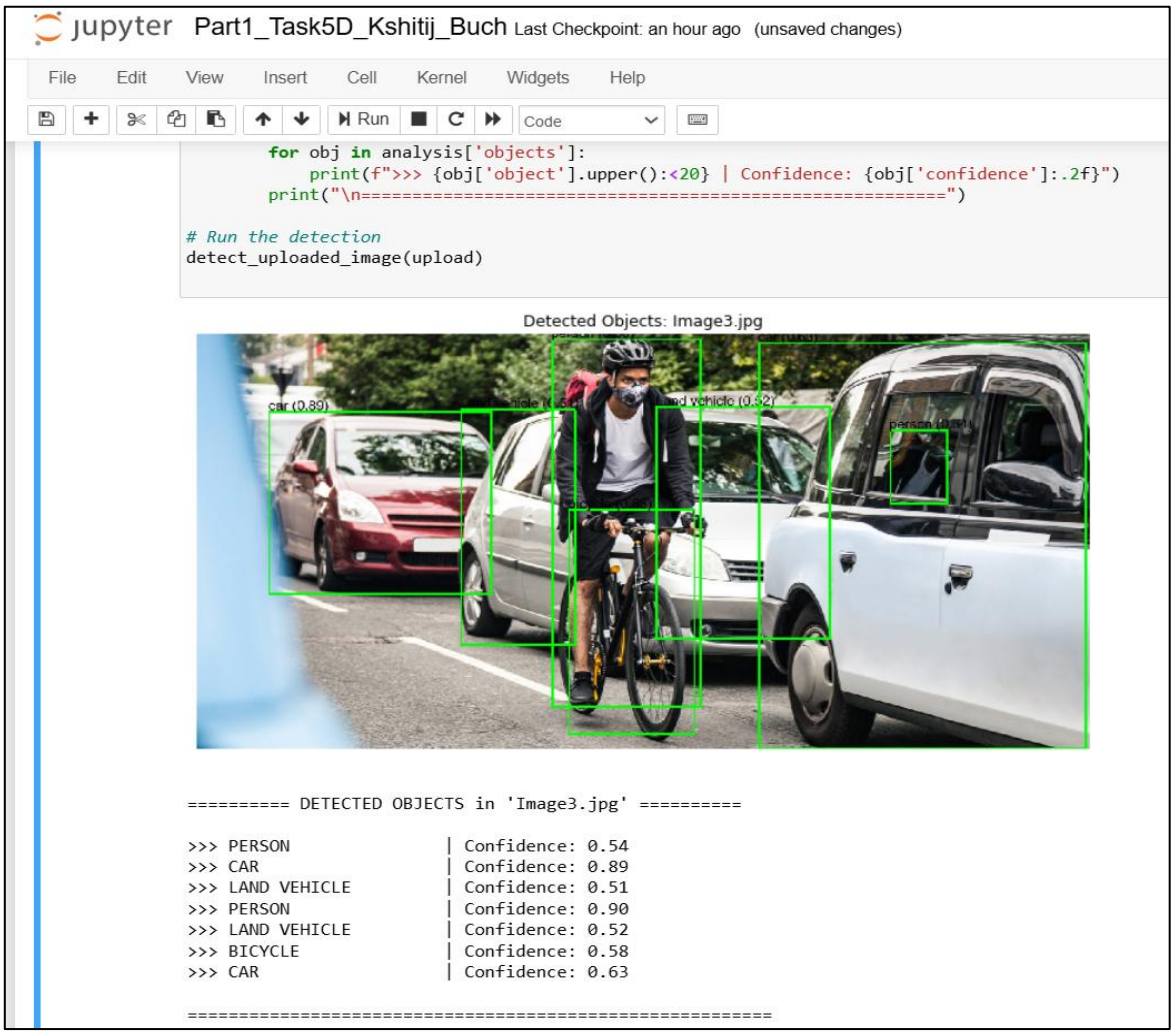


Fig5: Detected Objects With Bounding Boxes and Classification Labels & Their Confidence Scores

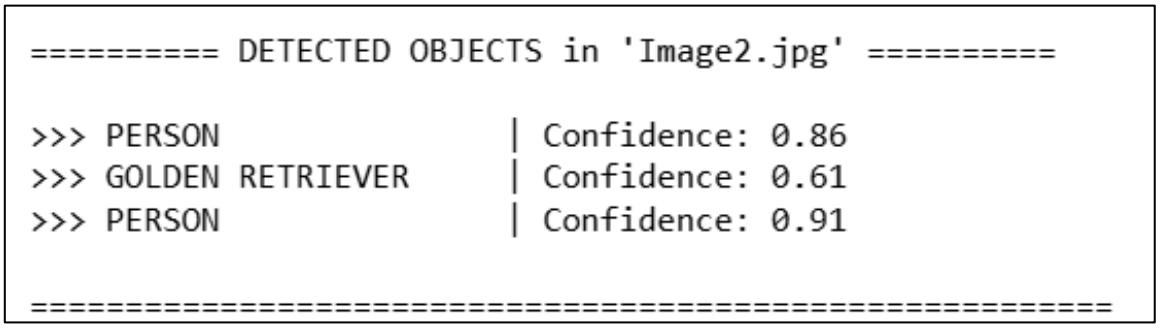


Fig6: Terminal Output Displaying Object Classification Labels And Their Confidence Scores

```
Detected Object Coordinates:
1. PERSON – Confidence: 0.86
   Bounding Box → X: 128, Y: 113, Width: 160, Height: 389

2. GOLDEN RETRIEVER – Confidence: 0.61
   Bounding Box → X: 210, Y: 183, Width: 200, Height: 310

3. PERSON – Confidence: 0.91
   Bounding Box → X: 323, Y: 31, Width: 225, Height: 447
```

Fig7: Object Coordinates Detected and Displayed

```
In [69]: import json

         # Display the JSON response
         print(f"JSON Output for image:{image_name}")
         print(json.dumps(analysis, indent=4))

JSON Output for image:Image2.jpg
{
    "objects": [
        {
            "rectangle": {
                "x": 128,
                "y": 113,
                "w": 160,
                "h": 389
            },
            "object": "person",
            "confidence": 0.857
        },
        {
            "rectangle": {
                "x": 210,
                "y": 183,
                "w": 200,
                "h": 310
            },
            "object": "golden retriever",
            "confidence": 0.606,
            "parent": {
                "object": "retriever",
                "confidence": 0.666,
                "parent": {
                    "object": "dog",
                    "confidence": 0.915,
                    "parent": {
                        "object": "mammal",
                        "confidence": 0.918,
                        "parent": {
                            "object": "animal",
                            "confidence": 0.918
```

Fig8: Sample JSON Output From Jupyter Notebook

6. Conclusion

This task successfully demonstrates object detection in images using the Azure Computer Vision **API**. It lays the foundation for tracking these detected objects across frames, which will be addressed in Part 2.

7. References

- Microsoft. 2024. *Computer Vision documentation - Azure AI services*. [online] Available at: https://learn.microsoft.com/en-us/azure/cognitive-services/computer-vision/ [Accessed 20 Mar 2025].
- Microsoft. 2024. *Vision Studio*. [online] Available at: https://portal.azure.com/ [Accessed 21 Mar 2025].
- Matplotlib. 2024. *Matplotlib: Visualization with Python*. [online] Available at: https://matplotlib.org/ [Accessed 22 Mar 2025].
- Python Software Foundation. 2024. *Pillow (PIL Fork) Documentation*. [online] Available at: https://pillow.readthedocs.io/ [Accessed 22 Mar 2025].
- Python Software Foundation. 2024. *Python Imaging Library (Pillow)*. [online] Available at: https://pillow.readthedocs.io/ [Accessed 23 Mar 2025].

8. Appendix A – Code Explanation

This appendix provides a concise breakdown of the Python code used for object detection in Part 1 of Task 5D using Azure Computer Vision API. Each step is explained to reflect the underlying logic and integration with cloud services.

1. Import Required Libraries

```
In [1]: import requests
        from PIL import Image, ImageDraw, ImageFont
        from io import BytesIO
        import matplotlib.pyplot as plt
        import ipywidgets as widgets
        from IPython.display import display
```

These libraries are essential for:

- Sending HTTP requests to Azure's REST API (requests)
- Reading, modifying, and annotating images (PIL)
- Visualizing results in the notebook (matplotlib)
- Uploading user-selected files in Jupyter Notebook (ipywidgets)

---

2. Uploading Image via Widget & Then Displaying It.

```
In [2]: # Create a file upload widget
        upload = widgets.FileUpload(accept='image/*', multiple=False)
        display(upload)
```

Upload (1)

A file upload widget is created to allow users to select an image interactively. Only one image is accepted at a time from the user.

---

3. Image Detection Function

```
In [3]: def detect_uploaded_image(upload_widget):
```

...

This function encapsulates the logic to:

- Extract the uploaded image
- Send it to Azure Computer Vision API
- Receive and parse the JSON response
- Annotate the image with detected objects and their bounding boxes
- Print summary information and bounding box dimensions

4. API Call to Azure Computer Vision

```
# Send request to Azure Vision API
headers = {
    "Ocp-Apim-Subscription-Key": subscription_key,
    "Content-Type": "application/octet-stream"
}
response = requests.post(analyze_url, headers=headers, data=image_bytes)
response.raise_for_status()
global analysis
analysis = response.json()
```

The image content is sent as a binary stream (octet-stream) using a POST request. A valid subscription key and endpoint URL are required to authenticate the call.

5. Load Image and Draw Bounding Boxes

```
# Load and draw on the image
image = Image.open(BytesIO(image_bytes)).convert("RGB")
draw = ImageDraw.Draw(image)

# Try to use a larger font
try:
    font = ImageFont.truetype("arial.ttf", size=26)  # Change size as needed
except IOError:
    font = ImageFont.load_default()  # Fallback font

for obj in analysis['objects']:
    rect = obj['rectangle']
    label = obj['object']
    conf = obj['confidence']
    x, y, w, h = rect['x'], rect['y'], rect['w'], rect['h']
    draw.rectangle([(x, y), (x + w, y + h)], outline="lime", width=4)
    draw.text((x, y - 25), f"{label} ({conf:.2f})", fill="black", font=font)
```

The image is converted into a PIL object for modification. Bounding boxes are drawn using coordinates provided in the API response. Labels and confidence scores are displayed above each object using an enhanced font size.

6. Display Annotated Image

```
# Display image with bounding boxes
plt.figure(figsize=(12, 10))
plt.imshow(image)
plt.axis("off")
plt.title(f"Detected Objects: {image_name}")
plt.show()
```

The processed image is displayed using matplotlib, without axes, and titled with the uploaded image name.

---

7. Print Object Labels and Confidence Scores

```
# Display text output in the notebook
print(f"\n========== DETECTED OBJECTS in '{image_name}' ==========\n")
for obj in analysis['objects']:
    print(f">>> {obj['object'].upper():<20} | Confidence: {obj['confidence']:.2f}")
print("\n============================================================")
```

This loop prints each detected object's classification label and associated confidence score.

---

8. Print Bounding Box Coordinates

```
In [8]: # Display object details with bounding box coordinates
        print("\nDetected Object Coordinates:")
        for idx, obj in enumerate(analysis['objects'], start=1):
            label = obj['object']
            conf = obj['confidence']
            rect = obj['rectangle']
            x, y, w, h = rect['x'], rect['y'], rect['w'], rect['h']

            print(f"{idx}. {label.upper()} - Confidence: {conf:.2f}")
            print(f"   Bounding Box → X: {x}, Y: {y}, Width: {w}, Height: {h}\n")
```

This part of the code iterates through all detected objects and displays their label, confidence, and precise bounding box coordinates (X, Y, Width, Height). This structured output helps understand the object's spatial position within the image and will be useful later for tracking in videos. Note idx is index of objects received from API call.

PART 2 : Object Tracking Using Azure Custom Vision API

1. Object Tracking

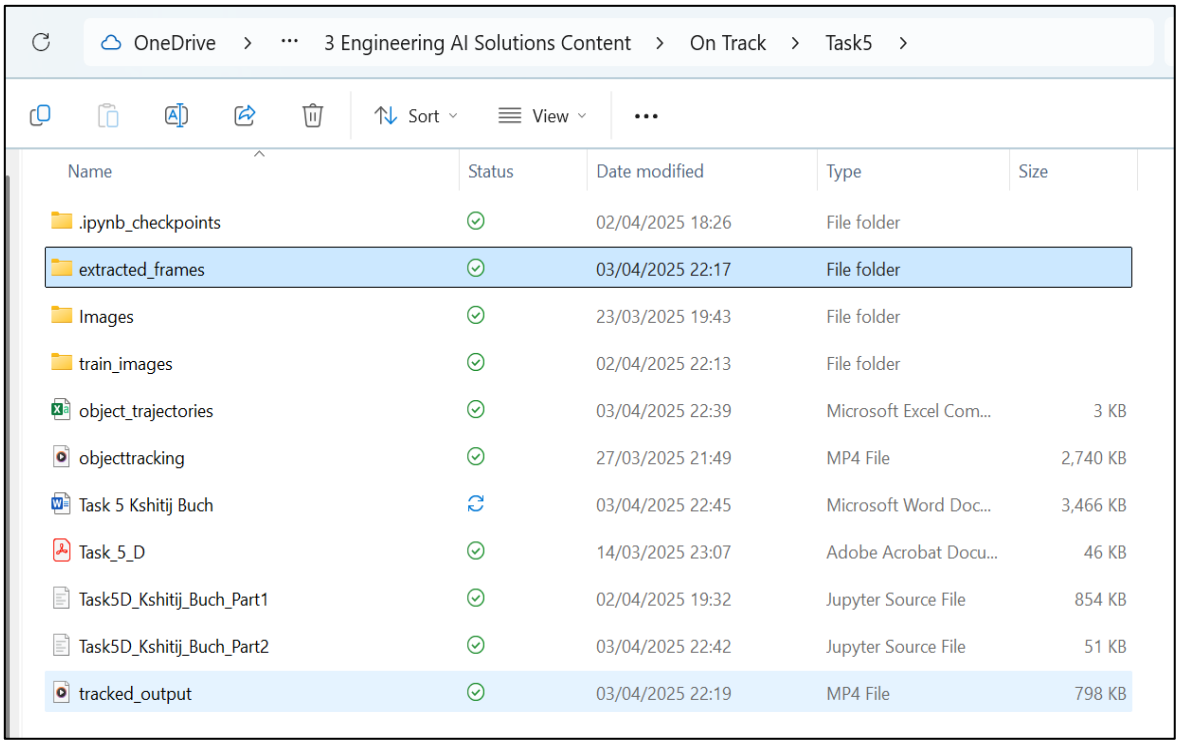
In this task, a custom object tracking pipeline is built using the Azure Custom Vision cloud service and Python SDK. The objective was to detect and track objects frame-by-frame in a video using a trained and published object detection model hosted on Azure's Custom Vision portal. The model was trained using the Azure Custom Vision UI, where I manually labelled the objects in a set of images to ensure the model's accuracy for the task. https://www.customvision.ai/projects/a0d9572d-1102-42b8-ab2e-6c544860ce3e#/performance

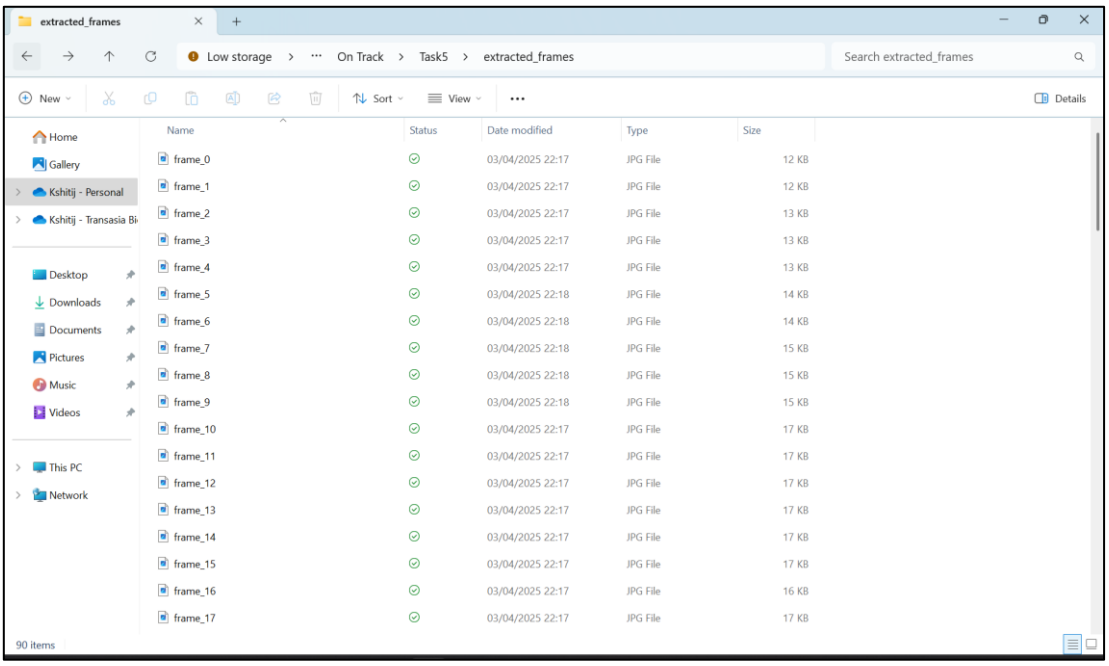
---

Steps Followed:

1. **Frame Extraction From Video:**

   - OpenCV's cv2.VideoCapture() is used to open and read the video named objecttracking.mp4.
   - Every second frame (alternate) was saved as an image inside a folder called extracted_frames.





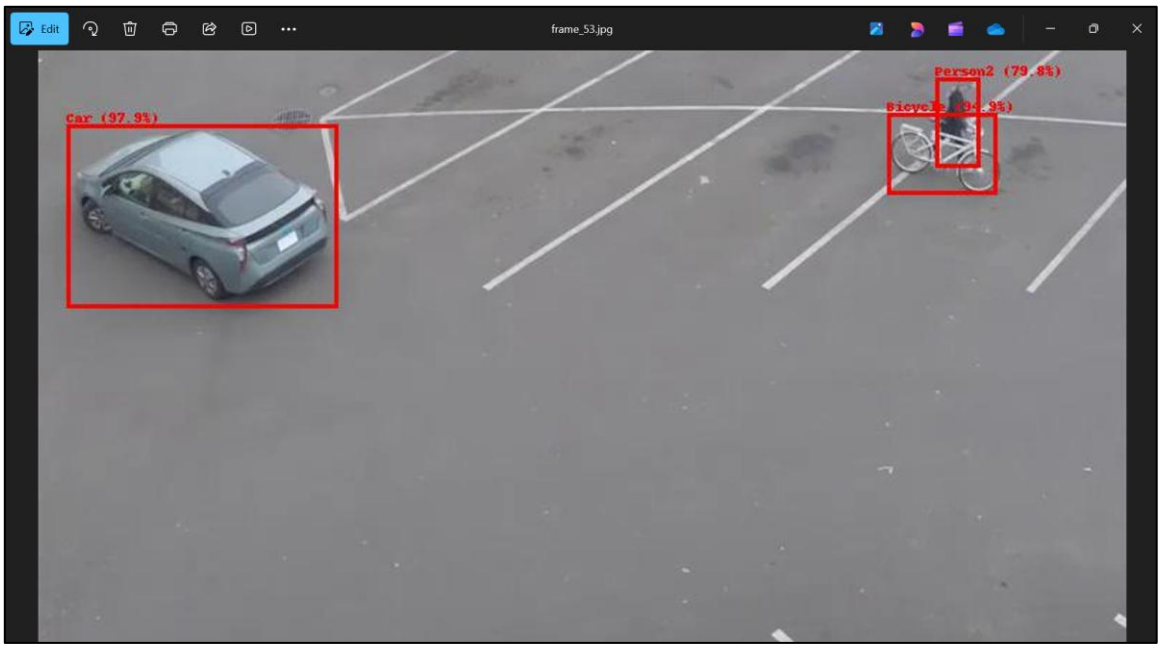
- This helped reduce the total number of images without missing too much movement.

2. **Model Prediction With Azure Custom Vision:**

- Each frame was read and sent to the Azure Custom Vision prediction endpoint using the CustomVisionPredictionClient.
- The model returned object predictions including the objects' tag name along with the bounding boxes coordinates and confidence or probability scores for those detected objects.

3. **Annotation:**

- Bounding boxes and labels were drawn on each frame using PIL.ImageDraw from PIL which is Python Image Library.
- The below image shows the red bounding box with the object name and its confidence score around the detected objects.
- The processed frames were saved back into the extracted_frames directory by overwriting the original frames. Below shown is a sample Annotated frame from the video.

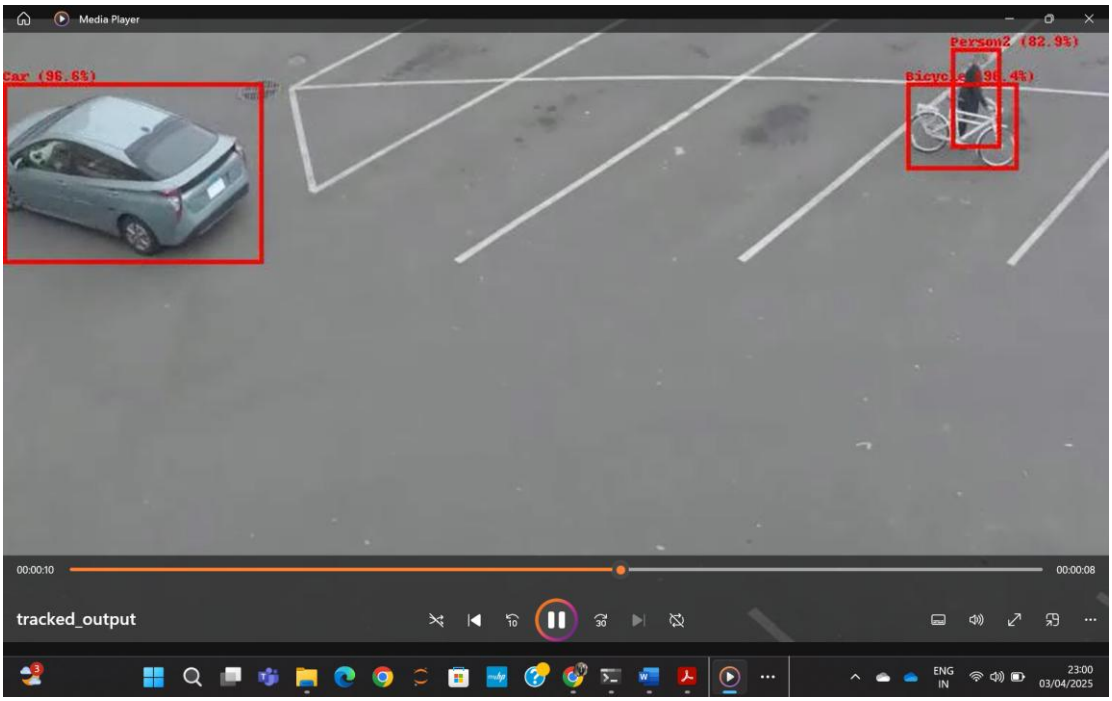


4. **Trajectory Tracking:**

- The center of each detected object (center_x and center_y) was calculated and for each tag (like Car, Person and Bicycle), these points were stored in a dictionary (trajectories) per object class.
- This allowed to map the movement of each object across multiple frames.

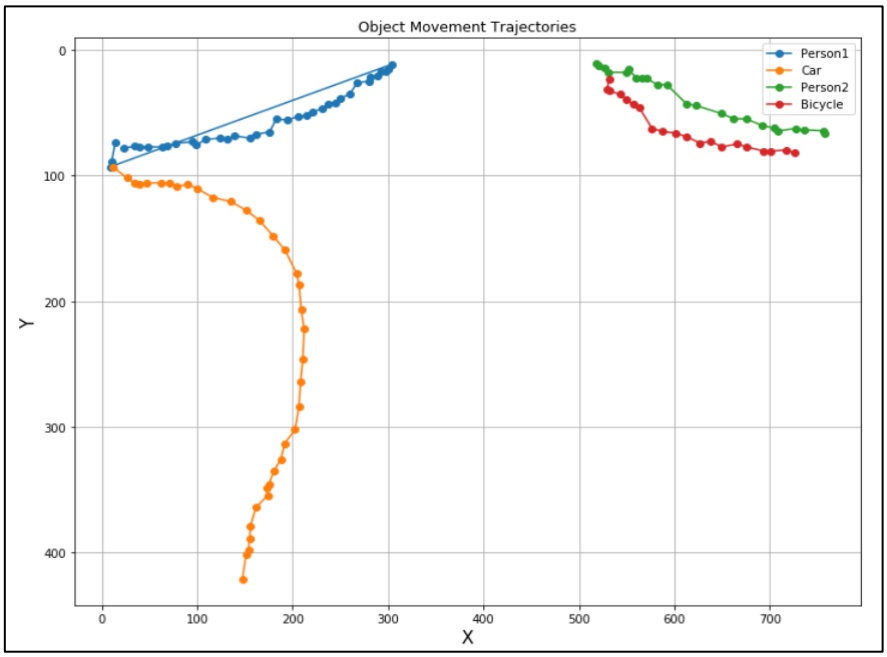
5. **Video Reconstruction:**

- The annotated frames were then joined back into a new video (tracked_output.mp4) using cv2.VideoWriter() to visualize the tracking output. Screenshot of that video playing is shown below



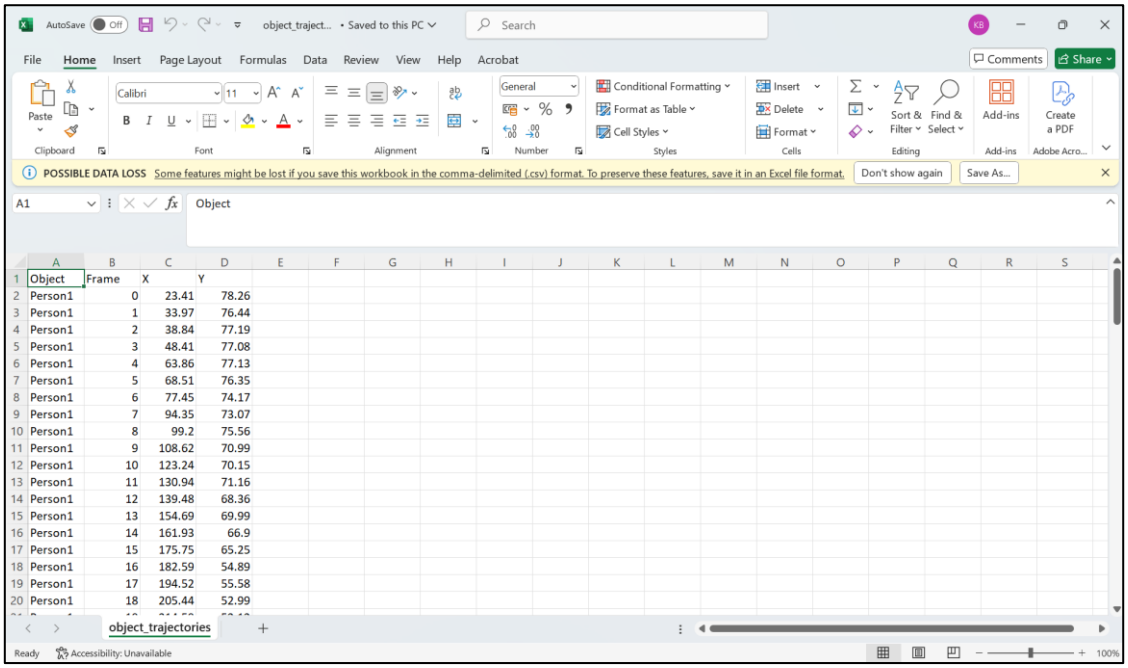
6. **Trajectory Visualization:**

- Using matplotlib, object movement trajectories were plotted based on the center points that were tracked across frames. The plot of which is shown below.
- .invert_yaxis() was used to match the screen's coordinate system because the one used by the openCV and PIL is having origin at top left of the screen.



7. **Exporting To CSV:**

- The trajectory data (object class/tag, frame index, X and Y coordinates i.e. center points) was saved into a CSV file named object_trajectories.csv for records and analysis purposes.

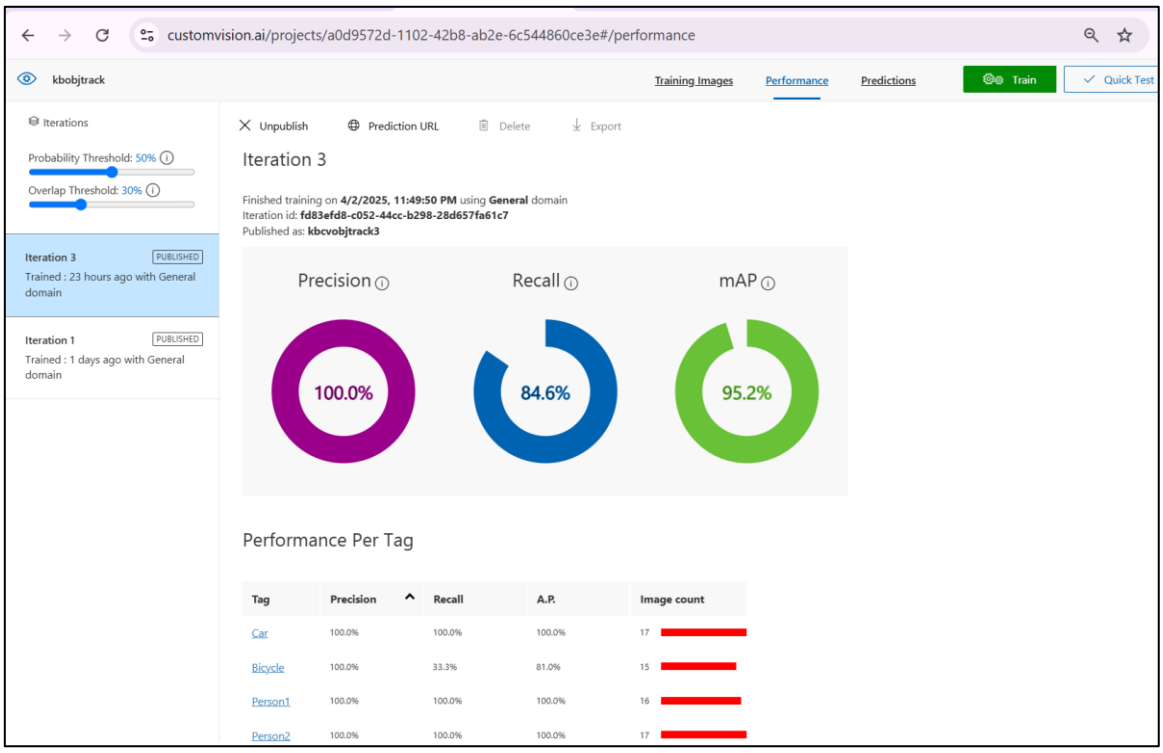


Object	Frame	X	Y
Person1	0	23.41	78.26
Person1	1	33.97	76.44
Person1	2	38.84	77.19
Person1	3	48.41	77.08
Person1	4	63.86	77.13
Person1	5	68.51	76.35
Person1	6	77.45	74.17
Person1	7	94.35	73.07
Person1	8	99.2	75.56
Person1	9	108.62	70.99
Person1	10	123.24	70.15
Person1	11	130.94	71.16
Person1	12	139.48	68.36
Person1	13	154.69	69.99
Person1	14	161.93	66.9
Person1	15	175.75	65.25
Person1	16	182.59	54.89
Person1	17	194.52	55.58
Person1	18	205.44	52.99

2. Performance Evaluation

Performance metrics were retrieved from the Azure Custom Vision portal after training and publishing the model:

Metric	Value	Meaning
Precision	100%	100% of the predicted objects were actually correct.
Recall	84.6%	84.6% of the actual objects were detected properly? Fewer missed objects.
mAP	95.2%	Combines both precision and recall across different thresholds. Very strong performance.



How well did it perform?

The object detection was found to be quite accurate. It was able to detect people, bicycles, and cars consistently even across a small number of frames. The bounding boxes were clean and stayed stable in most cases.

Limitations

- Sometimes the detection didn't work if the object was partially out of frame or blurred or occluded. Like the Bicycle when went out of the frame, was not detected for few frames which must have led to reduced Recall.
- Since it's a cloud API, it took time to send and receive results for each frame.

What can be improved?

- Add more varied training data, especially blurred and angled images.
- Use compact or real-time models if latency is an issue.
- For smoother object tracking, we can assign unique object IDs using additional logic.

3. Reporting and Documentation

Approach:

- Used the Azure Custom Vision SDK to automate the end-to-end video object tracking workflow.
- Programmatically sent each frame to the cloud-hosted model for prediction.
- Visualized both bounding boxes and object trajectories.

Findings:

- The published model was able to detect the trained object class with varying confidence levels.
- Trajectories effectively revealed the movement pattern of detected objects across frames.
- Some false negatives were observed, suggesting scope for further model tuning.

Conclusion:

- This task helped me understand how to work with Azure Custom Vision to build a complete object tracking pipeline using real video frames. I didn't need to manually annotate any data, which saved time. Although I faced a few challenges like setting up labelImg and latency of API responses, the final results were really satisfactory.
- The entire workflow – from frame extraction to prediction, drawing, and exporting – was done using Python, and I'm happy with the way it turned out.

4. References

- Microsoft. 2024. *Custom Vision documentation*. [online] Available at: https://learn.microsoft.com/en-us/azure/ai-services/custom-vision-service/ [Accessed 3 Apr 2025].
- OpenCV. 2024. *OpenCV: Open Source Computer Vision Library*. [online] Available at: https://docs.opencv.org/4.5.5/ [Accessed 3 Apr 2025].
- Pillow Developers. 2024. *Pillow (PIL Fork) Documentation*. [online] Available at: https://pillow.readthedocs.io/en/stable/ [Accessed 4 Apr 2025].
- Kili Technology. 2022. *Mean Average Precision (mAP): A complete guide*. [online] Available at: https://kili-technology.com/data-labeling/machine-learning/mean-average-precision-map-a-complete-guide [Accessed 4 Apr 2025].

5. Appendix A – Code Explanation

This appendix provides a concise breakdown of the Python code used for object tracking in Part 2 of Task 5D using Azure Custom Vision API. Each step is explained to reflect the underlying logic and integration with cloud services.

1. Import Required Libraries

These libraries are essential for:

- Capturing and processing video frames (cv2)
- Annotating images and calculating positions (PIL, ImageDraw)
- Communicating with Azure Custom Vision Prediction API (CustomVisionPredictionClient)
- Plotting trajectories using matplotlib
- Managing structured data like tag-wise object points (defaultdict)

```
In [10]: import cv2
         import os
         from azure.cognitiveservices.vision.customvision.prediction import CustomVisionPredictionClient
         from msrest.authentication import ApiKeyCredentials
         from PIL import Image, ImageDraw, ImageFont
         import matplotlib.pyplot as plt
         from collections import defaultdict
```

2. Extracting Frames from a Video

- Uses OpenCV to read the input video.
- Saves every 2nd frame to a folder to reduce redundancy and speed up processing.
- **Code**: cv2.VideoCapture(video_path) and cv2.imwrite()

```
In [11]: video_path = "objecttracking.mp4"
         extracted_frame_dir = "extracted_frames"
         os.makedirs(extracted_frame_dir, exist_ok=True)

         # start of capturing frames using cv2
         cap = cv2.VideoCapture(video_path)
         frame_interval = 2                                                    # Ext
         frame_id = 0                                                          # Is
         saved = 0

         while cap.isOpened():                                                 # Che
             ret, frame = cap.read()                                           # Rec
             if not ret:                                                       # ret
                 break
             if frame_id % frame_interval == 0:                                # Thi
                 filename = os.path.join(extracted_frame_dir, f"frame_{saved}.jpg") # Cre
                 cv2.imwrite(filename, frame)                                  # Thi
                 saved += 1                                                    # it
             frame_id += 1                                                     # Thi

         cap.release()                                                         # Thi
         print(f"Extracted {saved} frames into the directory '{extracted_frame_dir}/'")
```

3. Azure Custom Vision Prediction

- For each extracted frame, sends it to the prediction endpoint using the trained and published iteration.
- Gets prediction results with tag names, confidence scores, and bounding box dimensions.
- **Code**: predictor.detect_image (PROJECT_ID, PUBLISHED_NAME, image_data)

```
PUBLISHED_NAME = "kbcvobjtrack3"

credentials = ApiKeyCredentials(in_headers={"Prediction-Key": PREDICTION_KEY})
predictor = CustomVisionPredictionClient(PREDICTION_ENDPOINT, credentials)
```

Step3: Predict and annotate bounding boxes

```
In [13]: font = ImageFont.load_default()                                       # It loads a basic, b
         trajectories = defaultdict(list)                                      # Creates a dictionary

         for img_name in sorted(os.listdir(extracted_frame_dir)):              # Loops through each
             img_path = os.path.join(extracted_frame_dir, img_name)            # img_path gets the f
             with open(img_path, "rb") as image_data:                          # this file is opened
                 results = predictor.detect_image(PROJECT_ID, PUBLISHED_NAME, image_data)  # Sends the image to
```

4. Drawing Bounding Boxes

- Converts image to RGB.
- Uses bounding box coordinates to draw a rectangle and label.
- The center of the box is calculated and used to record movement.

```
    image = Image.open(img_path).convert("RGB")                                    # One Image is opened and i
    draw = ImageDraw.Draw(image)                                                   # This line readies to draw
    width, height = image.size                                                     # For the image that is ope

    for prediction in results.predictions:                                         # Loop through predictions
        if prediction.probability * 100 >= 50:                                     # Only images with confiden

# Below line of codes denormalises the values received from azure custom vision api by multiplying with actual
            left = prediction.bounding_box.left * width
            top = prediction.bounding_box.top * height
            w = prediction.bounding_box.width * width
            h = prediction.bounding_box.height * height
            center_x = left + w / 2                                                # This 2 line of code store
            center_y = top + h / 2

            tag = prediction.tag_name
            trajectories[tag].append((center_x, center_y))                         # Stores the center points

            draw.rectangle([left, top, left + w, top + h], outline="red", width=3)  # Draws a rectangle with r
            draw.text((left, top - 10), f"{tag} ({prediction.probability * 100:.1f}%)", fill="red", font=font)

    image.save(img_path)                                                           # replaces the annotated im
```

5. Tracking Object Positions

- Each tag (like ‘Person’ or ‘Car’) has a list of center coordinates stored.
- This helps in plotting the movement path (trajectory) of each object.

6. Reconstructing the Video

- Annotated frames are stitched back into a new video using OpenCV.
- Code: cv2.VideoWriter() with frame dimensions and FPS.

```
In [14]: frame_files = sorted([f for f in os.listdir(extracted_frame_dir) if f.endswith(".jpg")])
         first_frame = cv2.imread(os.path.join(extracted_frame_dir, frame_files[0]))
         h, w, _ = first_frame.shape
         video_out_path = "tracked_output.mp4"
         video = cv2.VideoWriter(video_out_path, cv2.VideoWriter_fourcc(*"mp4v"), 5, (w, h))

         for frame_file in frame_files:
             frame = cv2.imread(os.path.join(extracted_frame_dir, frame_file))
             video.write(frame)
         video.release()

         print("\n Object tracking video saved as:", video_out_path)
```

7. Trajectory Plotting

- Uses Matplotlib to plot movement lines per tag.
- Inverts the Y-axis to match image coordinate space.

```
In [19]: plt.figure(figsize=(10, 8))
         for tag, points in trajectories.items():
             xs, ys = zip(*points)
             plt.plot(xs, ys, marker='o', label=tag)

         plt.gca().invert_yaxis()
         plt.title("Object Movement Trajectories")
         plt.xlabel("X", fontsize = 15)
         plt.ylabel("Y", fontsize = 15)
         plt.legend()
         plt.grid(True)
         plt.tight_layout()
         plt.show()
```

8. Saving to CSV

- All object center points are written to object_trajectories.csv for potential analysis.

```
In [20]: import csv

         with open("object_trajectories.csv", mode="w", newline="") as file:
             writer = csv.writer(file)
             writer.writerow(["Object", "Frame", "X", "Y"])

             for tag, points in trajectories.items():
                 for frame_idx, (x, y) in enumerate(points):
                     writer.writerow([tag, frame_idx, round(x, 2), round(y, 2)])

         print("Trajectory data saved to 'object_trajectories.csv'")
```