Object detection using DOTA dataset and training using YOLOV8.

Submitted by: Ann Nibana Stephen Lal

` Course: EPPS6323(Knowledge Mining)

Professor**: Dr. Karl Ho**

Object detection using DOTA dataset and training using YOLOV8.

This project focuses on detecting airplanes in aerial imagery using a subset of the DOTA-v1.0 dataset and training a YOLOv8 model. The dataset was converted to COCO format, structured, and split into training and validation sets. After resolving label and compatibility issues, the model trained for 50 epochs chosen. Predictions were generated and visualized using a custom output directory, demonstrating object detection.

Workflow

The workflow included data preparation, format conversion, model training with YOLOv8s and performance evaluation.

1. **Data Preparation and Processing**

The dataset preparation began by acquiring the DOTA-v1.0 dataset, which includes high-resolution aerial images and corresponding labelTxt annotations. These annotations contain object class labels and oriented bounding box coordinates. From this large dataset, a subset of 40 images containing the class "airplane" was selected using a Python script to reduce computational load and streamline the model training process. This targeted selection ensured the model would focus solely on detecting airplanes, improving efficiency for a class-specific object detection task.

Once selected, the data needed to be converted into a format compatible with YOLOv8. Using the DOTA DevKit, the labelTxt annotations were transformed into the standard COCO format, producing a DOTA_1.0.json file containing image metadata, bounding boxes and class labels. The dataset was then split into two subsets: 30 images allocated for training (images/train) and

10 for validation (images/validation). COCO-style JSON files were manually or programmatically generated to align with these splits. This structured preparation ensured smooth downstream processing with YOLOv8s.

Folder structure

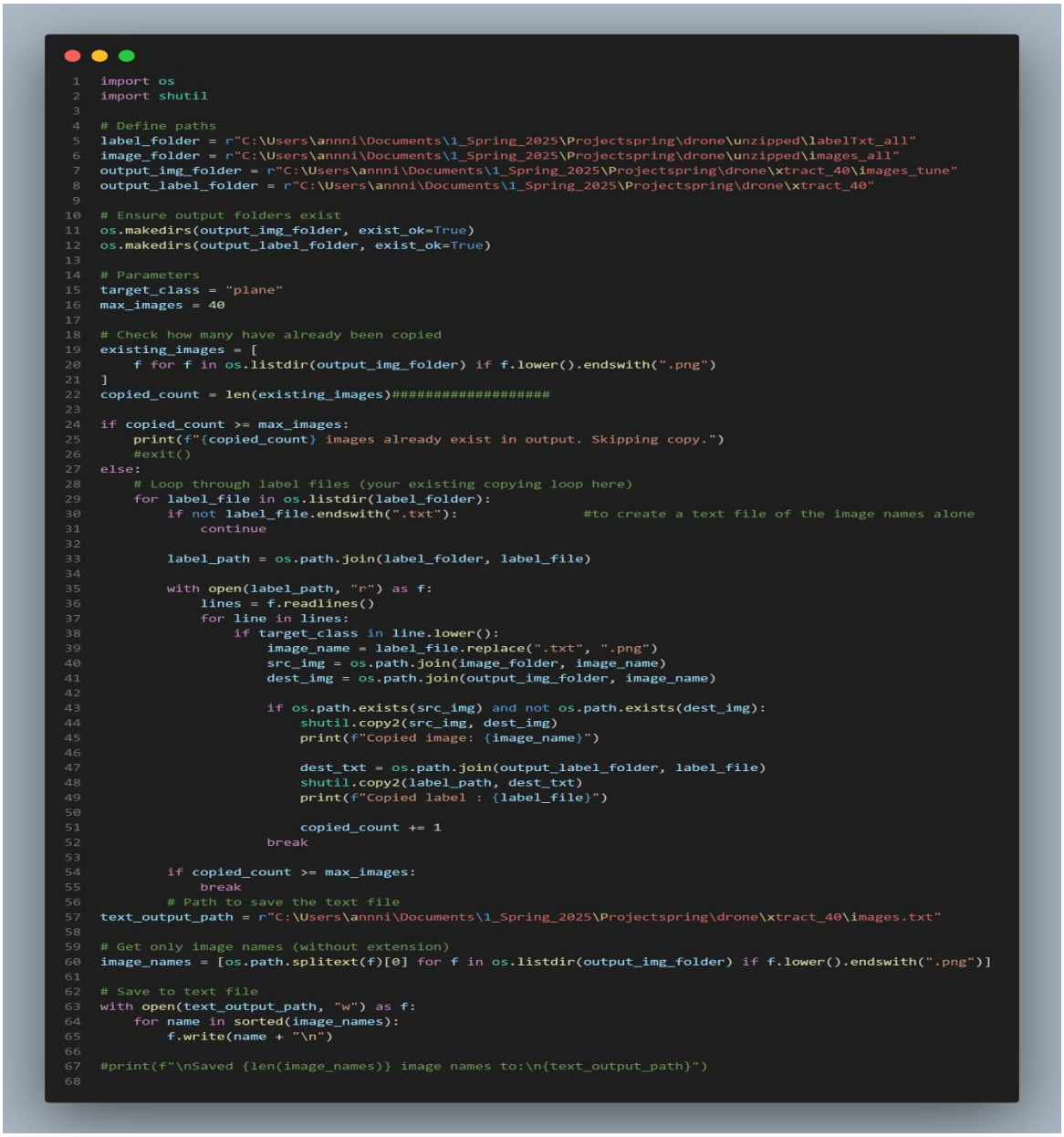
The folder structure is organized to align with YOLOv8's training requirements. The mini_train/ directory contains subfolders for images/ and labels/, each divided into train and val sets, separating training and validation data clearly. The annotations/ folder holds COCO-format JSON files, which are useful for conversions. A data.yaml file ties everything together by specifying image paths and class names, enabling model training.

mini_train/

├── images/
│ ├── train/ →Contains training images.
│ └── val/ →Contains validation images.
├── labels/
│ ├── train/ → YOLO-format label files for training images.
│ └── val/ →YOLO-format label files for validation images.
├── annotations/
│ ├── instances_train.json →COCO-style annotations for training.
│ └── instances_val.json →COCO-style annotations for validation.
│Created a data.yaml file

Image and label_txt Extraction

Python script used to extract exactly 40 images containing the class "plane" from a larger DOTA-style dataset.. The script loops through label files in the labelTxt_all folder, searching each for the presence of the word "plane". When a match is found, it copies the corresponding .png image from images_all and its .txt label file to the destination folders. Once 40 matching images and txt are copied, it stops.



```python
import os
import shutil

# Define paths
label_folder = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\unzipped\labelTxt_all"
image_folder = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\unzipped\images_all"
output_img_folder = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\xtract_40\images_tune"
output_label_folder = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\xtract_40"

# Ensure output folders exist
os.makedirs(output_img_folder, exist_ok=True)
os.makedirs(output_label_folder, exist_ok=True)

# Parameters
target_class = "plane"
max_images = 40

# Check how many have already been copied
existing_images = [
    f for f in os.listdir(output_img_folder) if f.lower().endswith(".png")
]
copied_count = len(existing_images)###################

if copied_count >= max_images:
    print(f"{copied_count} images already exist in output. Skipping copy.")
    #exit()
else:
    # Loop through label files (your existing copying loop here)
    for label_file in os.listdir(label_folder):
        if not label_file.endswith(".txt"):               #to create a text file of the image names alone
            continue

        label_path = os.path.join(label_folder, label_file)

        with open(label_path, "r") as f:
            lines = f.readlines()
            for line in lines:
                if target_class in line.lower():
                    image_name = label_file.replace(".txt", ".png")
                    src_img = os.path.join(image_folder, image_name)
                    dest_img = os.path.join(output_img_folder, image_name)

                    if os.path.exists(src_img) and not os.path.exists(dest_img):
                        shutil.copy2(src_img, dest_img)
                        print(f"Copied image: {image_name}")

                        dest_txt = os.path.join(output_label_folder, label_file)
                        shutil.copy2(label_path, dest_txt)
                        print(f"Copied label : {label_file}")

                        copied_count += 1
                    break

        if copied_count >= max_images:
            break
        # Path to save the text file
text_output_path = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\xtract_40\images.txt"

# Get only image names (without extension)
image_names = [os.path.splitext(f)[0] for f in os.listdir(output_img_folder) if f.lower().endswith(".png")]

# Save to text file
with open(text_output_path, "w") as f:
    for name in sorted(image_names):
        f.write(name + "\n")

#print(f"\nSaved {len(image_names)} image names to:\n{text_output_path}")

```

Splitting data for training and test .

Once the images and their annotations manually separated as 30 images for training, 10 for validation in subfolders named‘train’ and ‘val’ . Python script is made use of to split a COCO-format JSON file (DOTA_1.0.json) into two separate COCO-format JSON files: instances_train.json and instances_val.json saved under annotations folder, based on the images present in the train and val subfolders within the dataset directory (mini_train).

```python
import json
import os

# === Paths ===
base_dir = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\mini_train"
full_json_path = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\xtract_40\DOTA_1.0.json"
train_images_folder = os.path.join(base_dir, "images", "train")
val_images_folder = os.path.join(base_dir, "images", "val")
annotations_folder = os.path.join(base_dir, "annotations")

# Output files
train_json_output = os.path.join(annotations_folder, "instances_train.json")
val_json_output = os.path.join(annotations_folder, "instances_val.json")

#####safe EXIT
if os.path.exists(train_json_output) and os.path.exists(val_json_output):
    print("Split JSON files already exist. Skipping split process.")
else:
    # Run only if files don't exist
    extract_split(train_images_folder, train_json_output, "Training")
    extract_split(val_images_folder, val_json_output, "Validation")
    #########


# Make sure output folder exists
os.makedirs(annotations_folder, exist_ok=True)

# === Load original full JSON ===
with open(full_json_path, 'r') as f:
    coco_data = json.load(f)

# === Helper function to extract split ===
def extract_split(image_folder, output_json_path, split_name):
    image_files = {f for f in os.listdir(image_folder) if f.lower().endswith('.png')}

    # Filter images
    split_images = [img for img in coco_data['images'] if img['file_name'] in image_files]
    split_image_ids = {img['id'] for img in split_images}

    # Filter annotations
    split_annotations = [ann for ann in coco_data['annotations'] if ann['image_id'] in split_image_ids]

    # Assemble split JSON
    output = {
        'info': coco_data.get('info', {}),
        'licenses': coco_data.get('licenses', []),
        'images': split_images,
        'annotations': split_annotations,
        'categories': coco_data.get('categories', [])
    }

    # Save
    with open(output_json_path, 'w') as f:
        json.dump(output, f, indent=4)

    print(f" {split_name} JSON saved to: {output_json_path} ({len(split_images)} images, {len(split_annotations)} annotations)")

# === Run for both splits ===
extract_split(train_images_folder, train_json_output, "Training")
extract_split(val_images_folder, val_json_output, "Validation")

```

YOLO-based label format conversion

Taking into account the need for the COCO-format annotations from instances_train.json and instances_val.json to be converted into YOLO-format label files for training and validation sets, saving them as .txt files in labels/train and labels/val folders within the mini_train directory.The python script loads each JSON file, maps image IDs to filenames and dimensions and transforms COCO bounding box coordinates (x, y, width, height) into YOLO format (normalized center x, center y, width, height). For each image, it writes a corresponding .txt file containing the category ID and normalized bounding box coordinates.

```python
import os
import json

# === Paths ===
base_dir = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\mini_train"

# Define input COCO-style JSONs
splits = {
    "train": os.path.join(base_dir, "annotations", "instances_train.json"),
    "val": os.path.join(base_dir, "annotations", "instances_val.json")
}

# Output label folders
label_base = os.path.join(base_dir, "labels")
os.makedirs(label_base, exist_ok=True)

for split, json_path in splits.items():
    # Create label subfolder
    label_dir = os.path.join(label_base, split)
    os.makedirs(label_dir, exist_ok=True)

    # Load JSON
    with open(json_path, 'r') as f:
        data = json.load(f)

    # Build image_id to file name and size map
    image_info = {
        img["id"]: {
            "file_name": img["file_name"],
            "width": img["width"],
            "height": img["height"]
        } for img in data["images"]
    }

    # Group annotations by image
    annotations_by_image = {}
    for ann in data["annotations"]:
        img_id = ann["image_id"]
        annotations_by_image.setdefault(img_id, []).append(ann)

    # Process each image
    for img_id, anns in annotations_by_image.items():
        info = image_info[img_id]
        img_name = os.path.splitext(info["file_name"])[0]
        label_file = os.path.join(label_dir, f"{img_name}.txt")

        with open(label_file, "w") as f_out:
            for ann in anns:
                category_id = ann["category_id"]
                bbox = ann["bbox"]  # COCO format: [x, y, width, height]

                # Convert to YOLO format
                x_center = (bbox[0] + bbox[2] / 2) / info["width"]
                y_center = (bbox[1] + bbox[3] / 2) / info["height"]
                width = bbox[2] / info["width"]
                height = bbox[3] / info["height"]

                # YOLO format: <class_id> <x_center> <y_center> <width> <height>
                f_out.write(f"{category_id} {x_center:.6f} {y_center:.6f} {width:.6f} {height:.6f}\n")

    print(f"{split} labels saved to: {label_dir}")
```

Single class detection

Here as I am interested in detecting the presence of an object (aeroplane) rather than distinguishing between specific classes. The model can be trained in a way that all .txt label files in the labels/train and labels/val directories within the mini_train folder by replacing the first number (class ID) in each line with 0.

```python
import os

# === Folders ===
label_dirs = [
    r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\mini_train\labels\train",
    r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\mini_train\labels\val"
]

# === Replace class IDs with 0 ===
for label_dir in label_dirs:
    for filename in os.listdir(label_dir):
        if filename.endswith(".txt"):
            path = os.path.join(label_dir, filename)
            with open(path, "r") as f:
                lines = f.readlines()

            updated_lines = []
            for line in lines:
                parts = line.strip().split()
                if len(parts) >= 5:
                    parts[0] = "0"  # overwrite class ID
                    updated_lines.append(" ".join(parts) + "\n")

            with open(path, "w") as f:
                f.writelines(updated_lines)

print(" All class IDs changed to 0.")
```

Resizing images

Resizing is necessary because YOLO requires images to be a uniform size for compatibility during training and inference resizes images from the train and val folders within the mini_train/images

directory to a uniform size of 640x640 pixels, saving the resized images to corresponding train_resized and val_resized folders. Using the PIL library, pads it to the target size with a gray background (RGB: 114, 114, 114) to avoid distortion, then saves the result in the output directory.

```python
from PIL import Image, ImageOps
import os

# === Paths ===
base_dir = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\mini_train\images"
train_input_dir = os.path.join(base_dir, "train")
val_input_dir = os.path.join(base_dir, "val")

train_output_dir = os.path.join(base_dir, "train_resized")
val_output_dir = os.path.join(base_dir, "val_resized")

target_size = (640, 640)

# === Resize function with padding ===
def resize_images(input_dir, output_dir, set_name):
    os.makedirs(output_dir, exist_ok=True)
    count = 0

    for file in os.listdir(input_dir):
        if file.lower().endswith((".png", ".jpg", ".jpeg")):
            input_path = os.path.join(input_dir, file)
            output_path = os.path.join(output_dir, file)
            try:
                with Image.open(input_path) as img:
                    padded_img = ImageOps.pad(img, target_size, color=(114, 114, 114))
                    padded_img.save(output_path)
                    count += 1
            except Exception as e:
                print(f"  Failed to process {file}: {e}")

    print(f" {set_name} set: {count} images resized to {target_size} and saved to: {output_dir}")

# === Run for both sets ===
resize_images(train_input_dir, train_output_dir, "Training")
resize_images(val_input_dir, val_output_dir, "Validation")

```

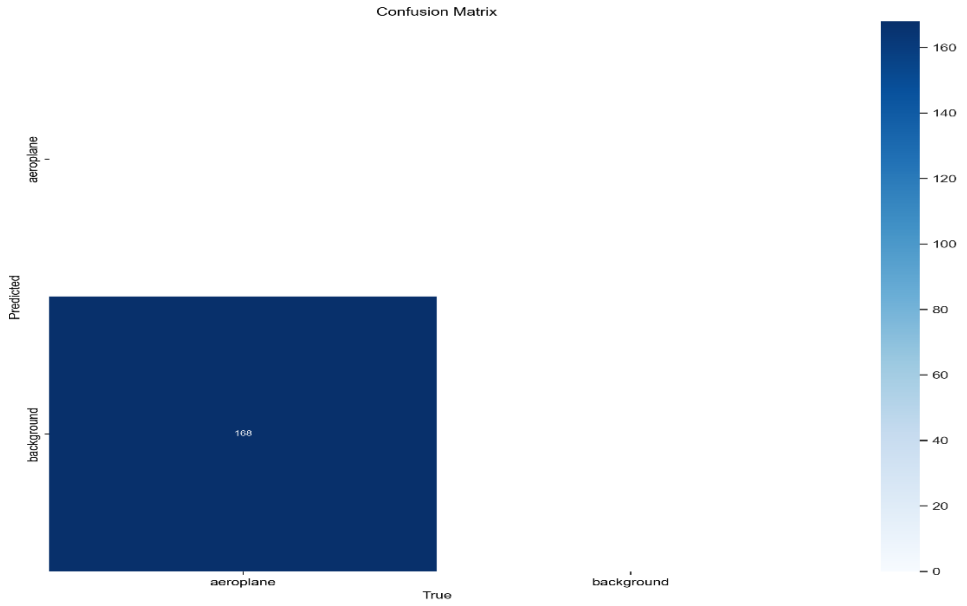
2. Set up and Training

The installation process for YOLOv8 involved setting up Python 3.9 and installing the Ultralytics library using pip install ultralytics, creating a compatible environment for object detection.

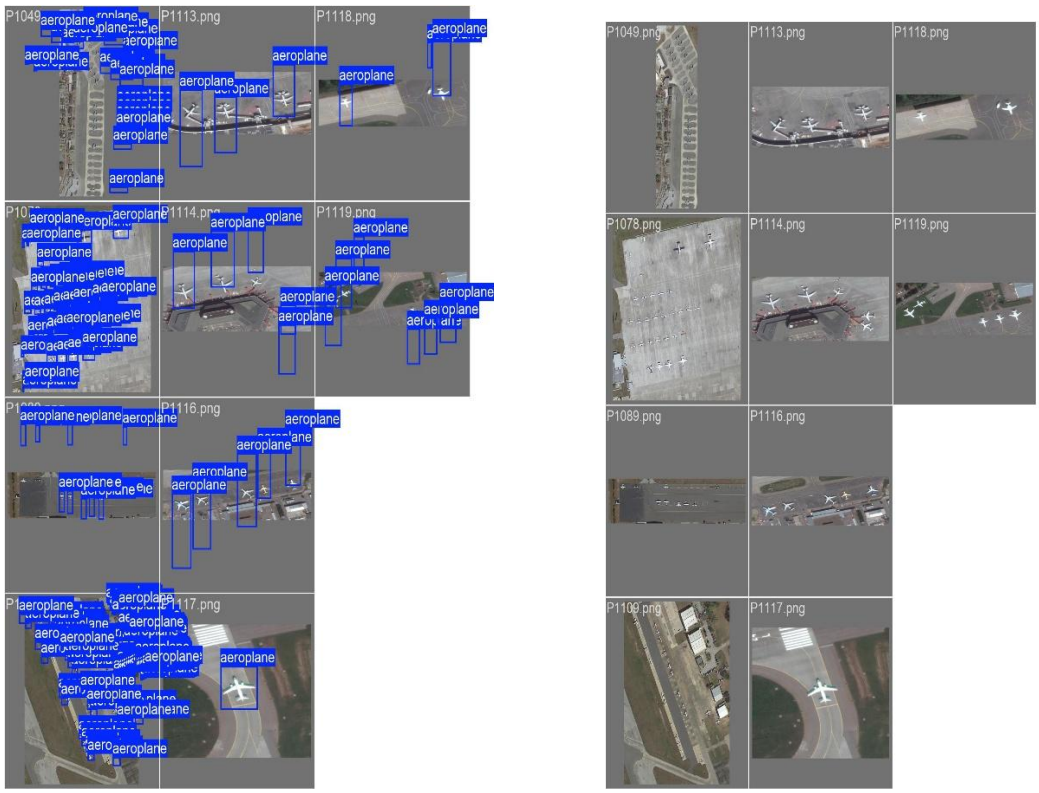
Training Epochs

During training, multiple epochs namely 10, 30 and 50 were carried out to see reduce in prediction error . The YOLOv8 framework monitors the validation performance after every epoch and saves the model state that performs best as best.pt. The model best.pt that resulted from epoch 50 was used for the detection of aeroplanes in new images.

Epoch 10



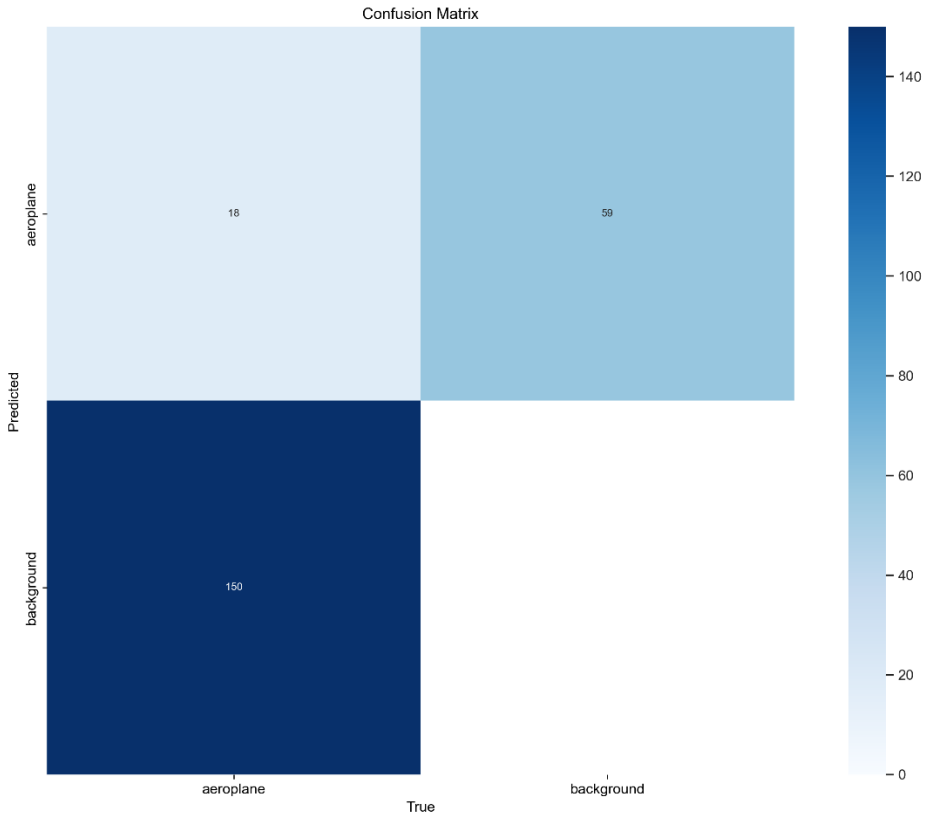
This confusion matrix shows that the model failed to detect any aeroplane instances, predicting all of them as background. All 168 true aeroplane labels were misclassified, indicating a 0% true positive rate.



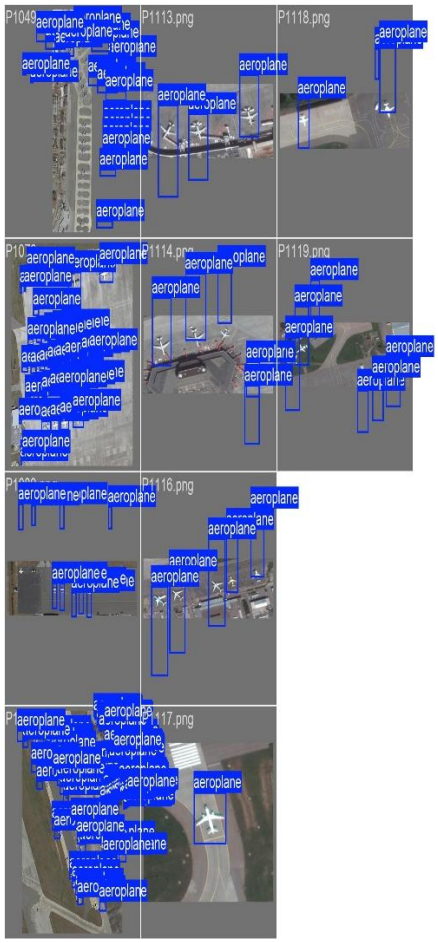
a *b*

The verification set(a) contains ground truth, serving as a baseline for assessing how well the model has learned to identify aeroplanes. The predicted output(b), on the other hand, shows the model's detection results, which here is none for he epoch10 iterations.

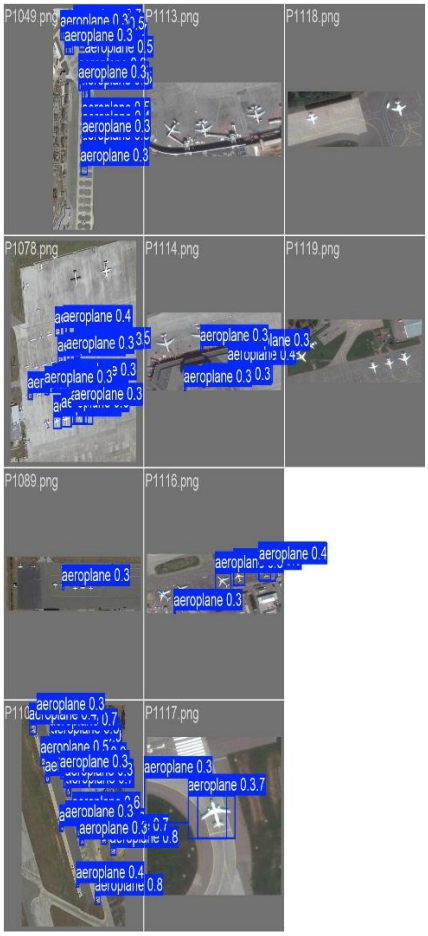
Epoch 30



This confusion matrix from epoch 30 shows moderate improvement in model performance. Out of the total aeroplane instances, 18 were correctly classified (true positives), while 150 were missed (false negatives), and 59 background instances were mistakenly identified as aeroplanes (false positives).



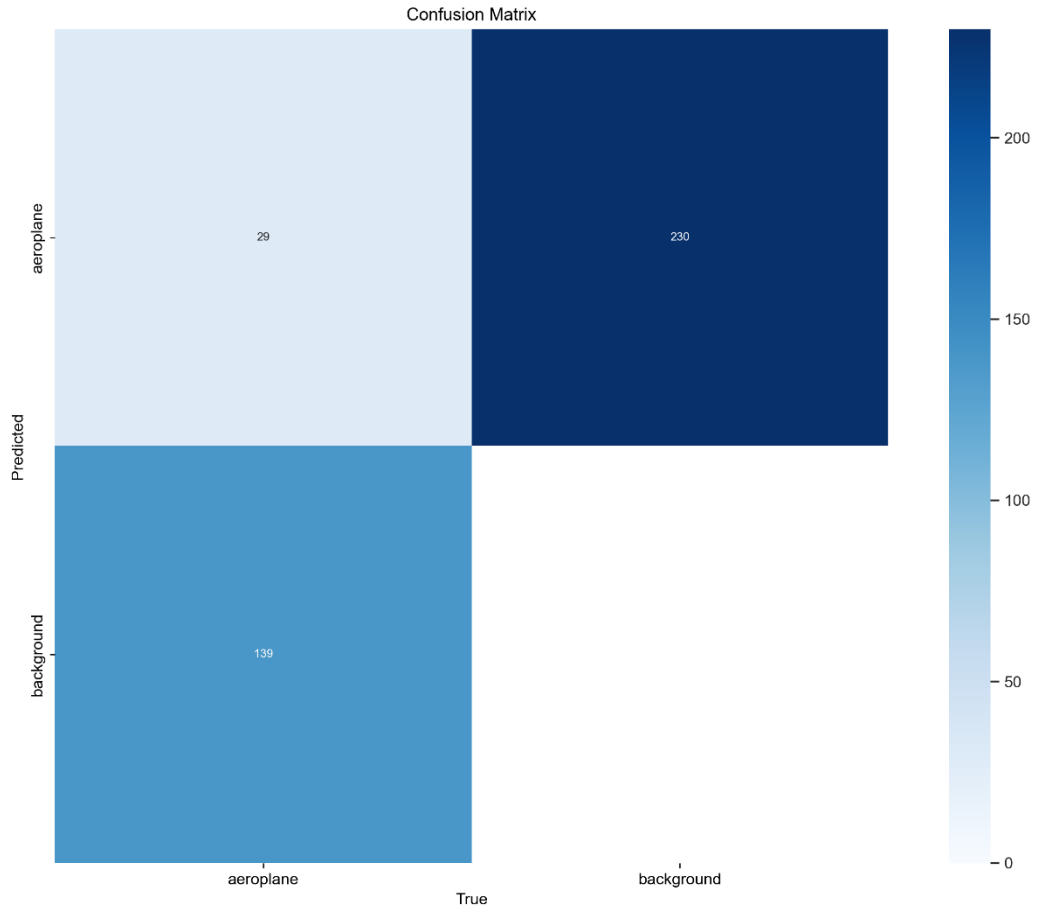
a



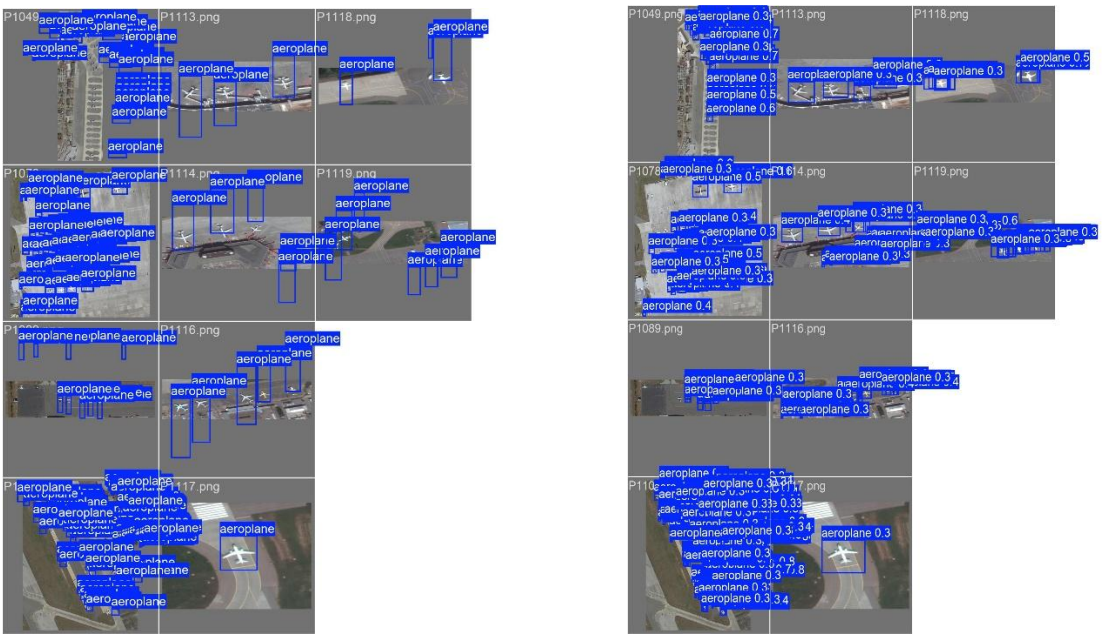
b

The first image shows the validation (a) with bounding boxes for airplanes. The second image displays the YOLOv8 model's predictions, where the model correctly identifies several airplanes with misses and false positives involved.

Epoch 50



The confusion matrix from epoch 50 shows a noticeable improvement in model performance. The model correctly identified 29 aeroplane instances (true positives), misclassified 139 as background (false negatives), and mistakenly predicted 230 background instances as aeroplanes (false positives). Compared to earlier epochs, the increase in true positives suggests better learning.



Validation vs predicted results for epoch 50 reveal improved model performance compared to earlier epochs. The predicted outputs (right) show more detection of aeroplanes but not entirely without misclassification.

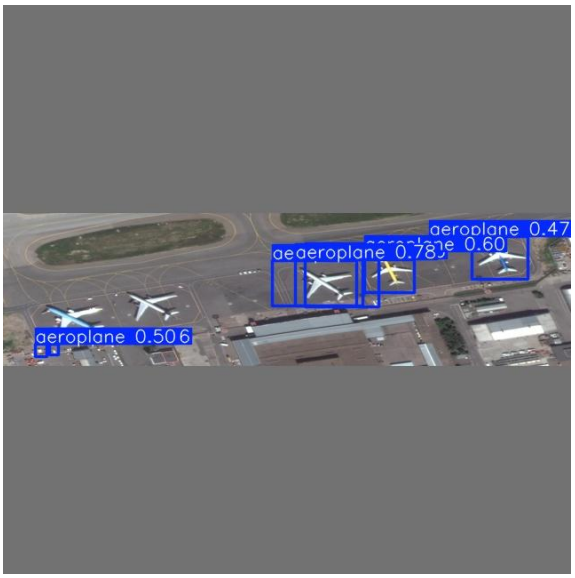
Prediction

In order to test the prediction, In this project, prediction was achieved by loading the trained model (specifically the best.pt file from 50 epochs trained on YOLOV8s version) and applying it to input images. A Python script was used to run inference on each image with a set confidence threshold and the model output annotated predictions and corresponding text files showing the detected bounding boxes and class labels.

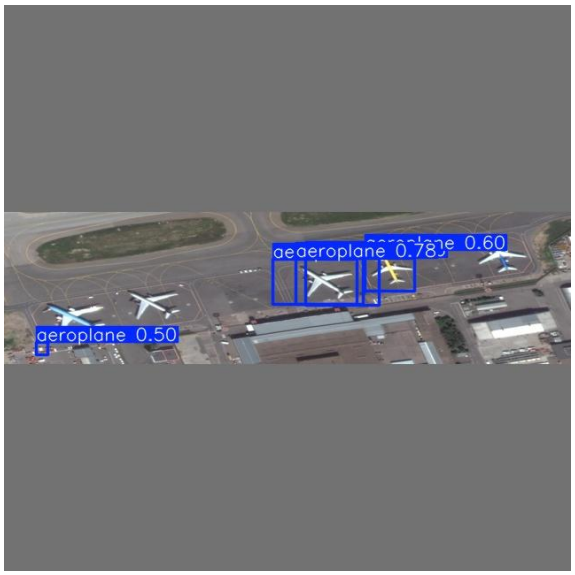
```python
from ultralytics import YOLO
import os

# Paths
input_dir = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\springproject\data\input_images"
output_dir = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\springproject\results_0.4"
model_path = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone\springproject\weights2\best.pt"

# Load model
model = YOLO(model_path)

# Run predictions on all images in input directory
for file in os.listdir(input_dir):
    if file.lower().endswith(('.png')):
        input_path = os.path.join(input_dir, file)
        model.predict(source=input_path, save=True, save_txt=True, conf=0.5, project=output_dir, name='predict_output', imgsz=640)
```

Confidence threshold variations

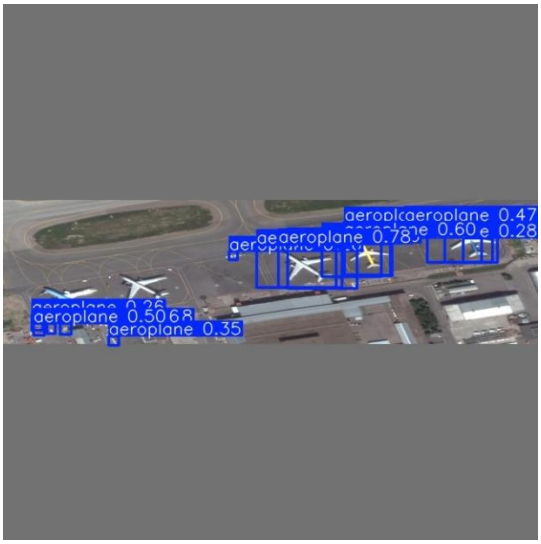
On running prediction on three same input images using different confidence thresholds (0.4, 0.5, and 0.25).



CF= 0.4



CF=0.5

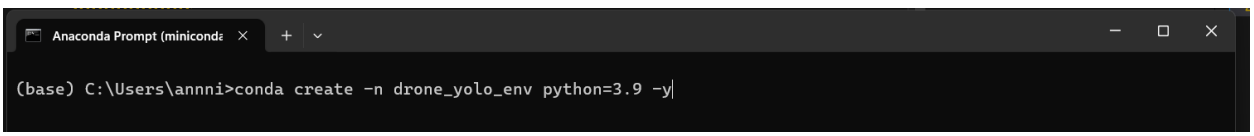


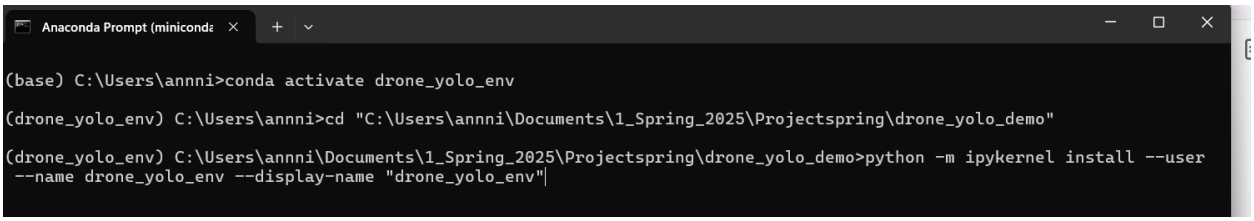
CF=0.25

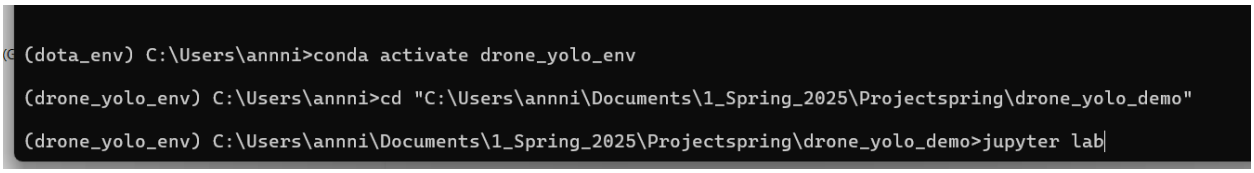
At 0.4, slightly more objects are detected with moderate confidence. At 0.5, only confident detections are shown, hence comparatively less airplanes detected.. At 0.25, even low-confidence predictions are included, increasing the number of boxes but also potentially introducing more false detections. This illustrates how lowering the threshold increases sensitivity at the cost of precision.

3. Interacting environment

To set up an interactive YOLOv8 environment, a Conda environment named drone_yolo_env was created with Python 3.9 and all necessary packages such as ultralytics, jupyterlab, and ipython were installed. The environment was linked to Jupyter using ipykernel to enable kernel switching. Within JupyterLab, a notebook (demo.ipynb) was used to load and visualize drone images, employing IPython.display.Image to render the predictions interactively, supporting quick testing andvalidation.



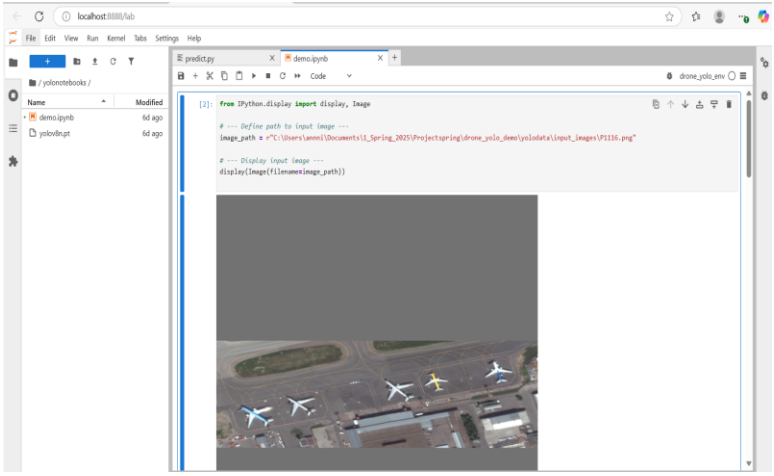




Activating the drone_yolo_env Conda environment and navigating to the YOLO demo project directory. Enables launching JupyterLab to run and interact with the project notebooks.

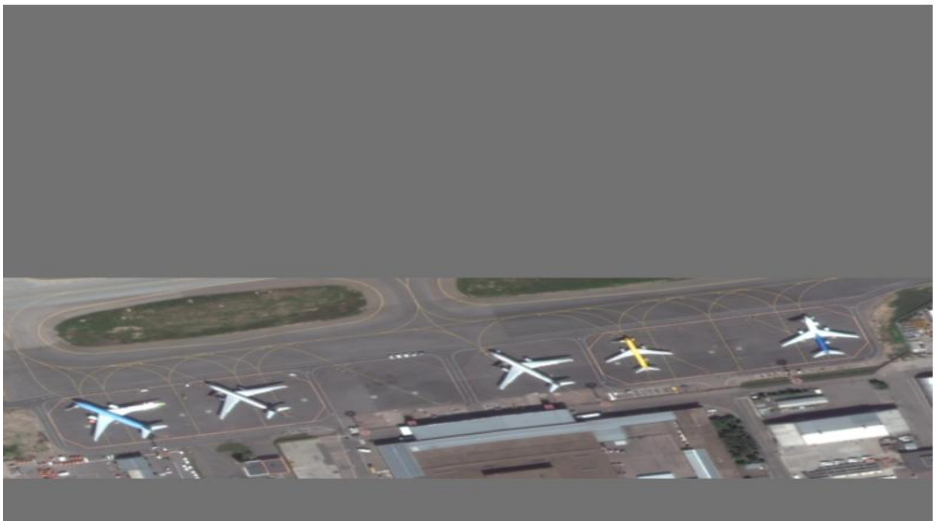
Verification and Display

In this JupyterLab environment, a notebook named demo.ipynb is used to visually inspect an input image before making predictions. The script utilizes IPython's display and Image modules to define the image path (P1116.png) and render it inline within the notebook interface.



Loads and displays a sample input image using the IPython.display module in JupyterLab. By specifying the file path of the image (P1116.png), the script renders the aerial image inline, allowing the user to visually confirm the image content before running detection.

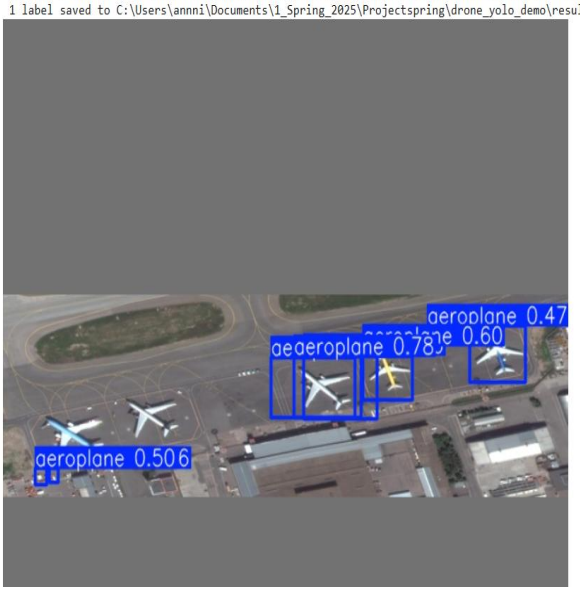
```
from IPython.display import display, Image

# --- Define path to input image ---
image_path = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone_yolo_demo\yolodata\input_images\P1116.png"

# --- Display input image ---
display(Image(filename=image_path))
```



Cell2 performs object detection using a YOLOv8 model within a Jupyter notebook. It defines the model and output paths, removes any existing output folder, loads the trained model (best.pt), and runs predictions on a specified image with a confidence threshold of 0.45. Finally, it displays the resulting annotated image inline using the IPython.display module, enabling easy visualization of the detected objects.

```python
import os
import shutil
from ultralytics import YOLO
from IPython.display import display, Image

# --- Define paths ---
model_path = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone_yolo_demo\weights\best.pt"
output_dir = r"C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone_yolo_demo\results"
predict_name = "predict_output2"
predict_folder = os.path.join(output_dir, predict_name)

# --- Remove old output folder if it exists ---
if os.path.exists(predict_folder):
    shutil.rmtree(predict_folder)

# --- Load YOLOv8 model ---
model = YOLO(model_path)

# --- Run prediction ---
results = model.predict(
    source=image_path,
    save=True,
    save_txt=True,
    project=output_dir,
    name=predict_name,
    imgsz=640,
    conf=0.45
)

# --- Display prediction image ---
predicted_path = os.path.join(results[0].save_dir, os.path.splitext(os.path.basename(image_path))[0] + ".jpg")
display(Image(filename=str(predicted_path)))
```

```
image 1/1 C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone_yolo_demo\yolodata\input_images\P1116.png: 640x640 7 aeroplanes, 1316.0ms
Speed: 11.4ms preprocess, 1316.0ms inference, 11.9ms postprocess per image at shape (1, 3, 640, 640)
Results saved to C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone_yolo_demo\results\predict_output2
1 label saved to C:\Users\annni\Documents\1_Spring_2025\Projectspring\drone_yolo_demo\results\predict_output2\labels
```



Conclusion

This project demonstrates the use of deeplearning model (YOLOV8) for object detection on a subset of the DOTA-v1.0 dataset, specifically targeting aeroplane detection. The workflow involved dataset preparation, format conversion to COCO, training the model over 50 epochs, and

evaluating predictions using visual outputs and confusion matrices. An interactive JupyterLab environment was created to support testing and visualization. While the model showed progressive improvement, the project also leaves rom for future enhancements.

Improvements and potential application

To improve this project, incorporating more training images would enhance model generalization. While YOLOv8s was used, experimenting with YOLOv8m can yield better. Increasing training epochs could further refine performance. Adjusting the confidence threshold allows for better precision-recall balance. Extracting object coordinates and overlaying them on a map could provide spatial insights, enabling real-world applications like infrastructure monitoring or geolocation tagging.

A potential application is collecting drone images, just as YOLOv8 was used to detect aeroplanes from imagery, employ similar approach to identify and analyze electric poles. By detecting poles and examining the geometry of their bounding boxes or fitted lines, it becomes possible to estimate their slant or tilt angles. This transformation of visual data into quantitative knowledge enables the assessment of pole alignment, valuable in monitoring hazards and planning maintenance.

References

Fei Feng, Yu Hu et al.*Improved YOLOv8 algorithms for small object detection in aerial imagery*. https://doi.org/10.1016/j.jksuci.2024.102113

Jocher, G., et al. (2023). YOLOv8 by Ultralytics, GitHub Repository: https://github.com/ultralytics/ultralytics/issues/8837

Jun Deng et al (2020), *A review of research on object detection based on deep learning.* J. Phys.: Conf. Ser. 1684 012028.

Solawetz, J. Francesco. 2023. *"What is YOLOv8? The Ultimate Guide."* Blog post. https://blog.roboflow.com/what-is-yolov8/

DOTA-v2 dataset for oriented bounding boxes (OBB). Ultralytics. https://docs.ultralytics.com/datasets/obb/dota-v2/

Tsung-Yi Lin Michael Maire (2015)*Microsoft COCO: Common Objects in Context* *https://arxiv.org/pdf/1405.0312*

Mupparaju Sohan,Thotakura Sai Ram et al(2024).*A Review on YOLOv8 and Its Advancements* 10.1007/978-981-99-7962-2_39

In addition to these references, I have used **ChatGPT** and **Grok** as AI assistants to help with my project. ChatGPT was used for writing and debugging Python scripts_for tasks namely extracting images and label txt, converting annotations to COCO format and data dimension correction. I used **Grok** for quick summaries to keep track of the progress of the project in between different stages.