COMP-10199-01 - Applied Research

Airplane Identification Research Project

Final Project Report

By: Armand Amores | Braeden Lyman | Cooper St.Martin | Nathan Ball

1. Abstract

Our Proposal

During World War II, the Royal Air Force (RAF) employed innovative training methods, such as silhouette playing cards, to improve aircraft recognition among personnel. Inspired by this historical approach and in collaboration with the National Air Force Museum of Canada (NAFMC), our project leverages modern artificial intelligence (AI) to identify aircraft from digitized photographs, enhancing public access to museum collections. Previous research teams made significant strides using YOLOv8 for aircraft detection, including generating synthetic training data from 3D models in Blender.

Building on this foundation, our work focuses on advancing the model's accuracy by transitioning to YOLOv11, refining classification by engine type (single-engine, multi-engine, jet, biplane), and tagging specific WWII-era aircraft from the NAFMC catalog. Using Roboflow for annotation, we expanded our dataset to over 1,000 labeled images, optimizing training parameters and augmentations to improve performance.

Technical Summary

Our technical implementation focused on enhancing the existing framework through several key improvements. We transitioned from YOLOv8 to the more advanced YOLOv11n architecture to achieve better detection accuracy, particularly in challenging scenarios with cluttered backgrounds. The dataset was significantly expanded to over 1,000 meticulously annotated images. Using Roboflow's annotation tools, we developed a sophisticated classification system that first identifies engine type (single-engine, multi-engine, jet, or biplane) before determining the specific aircraft model. The training process was optimized through systematic experimentation with hyperparameters, data augmentation techniques, and extended training epochs to address performance issues and improve detection reliability.

Challenges

The project encountered several significant technical hurdles during development. Setting up the GPU training environment proved particularly challenging, requiring careful configuration of CUDA and PyTorch compatibility along with proper installation of the Ultralytics package. Data quality emerged as another critical concern, as we needed to ensure our training set was both diverse and accurately labeled to prevent model bias. Performance issues manifested in choppy video inference, which we addressed through a combination of dataset expansion and parameter optimization. These challenges required iterative problem-solving and continuous refinement of our approach.

Approach

Our methodology followed a structured two-phase development process. During the first half of the semester, we conducted a thorough review of previous work, originally used YOLOv8 as our target model architecture, and focused on building a robust dataset through extensive image collection and annotation. The second half of the semester was dedicated performance optimization, where we expanded the training dataset, fine-tuned model parameters, and reorganized the project's GitHub repository for better maintainability. We also upgraded YOLOv8 to YOLOv11. To overcome data limitations, we implemented a comprehensive scraping strategy using Bing image search (custom bing scraper script) and Google image search (custom google scraper script) while maintaining strict quality control over our labeling process. This systematic approach allowed us to progressively enhance the model's accuracy while creating a foundation for future improvements and extensions to the project.

Conclusion

By combining historical inspiration with modern computer vision techniques, this project successfully bridges the gap between World War II-era training methods and contemporary AI applications. The developed solution not only serves as a valuable tool for the National Air Force Museum of Canada but also demonstrates the potential of object detection technology in cultural heritage preservation. Through overcoming numerous technical challenges, we've created a scalable system that accurately identifies historical aircraft while providing a framework for future enhancements in museum collection accessibility.

2. Activities & Progress

Midterm Report:

- Researched how yolo works
- Experimented with detecting first an airplane on images
- After going through tutorials and with this knowledge we reviewed previous groups work
- Leveraged tools from previous work
    - Web scraper (to scrape WWII planes instead)
    - Utilized models they trained to test how well it performs detecting multi-plane, single-engine, jet planes, etc.
- Utilized roboflow (a computer vision platform) that allows us to upload, annotate, organize, and classify data.
- Trained our Model based off these images of annotated data
- Ran an experiment by scraping 5 models of WWII planes off of bing search results.
- Uploaded and annotated that data and trained a YOLO v8 model with this data set.
- Tested results on images and video with great results
- Still needs work however. (We need more data)

Final Report Changes:

- Assigned each other images to annotate(200+) within Roboflow to increase the size of the dataset.
- Retrain model with new dataset
- Cleanup Github repo
- Prepare compatible VENV environment

3. Challenges and Solutions

Technical

- Hardware Limitations (CPU vs. GPU) and Virtual Environment (VENV)

  Issue: Without proper initialization you could be running a very poorly optimized environment for training models especially when using large datasets. To optimize performance, the virtual environment must be set correctly and your hardware should be compatible with the configurations set up. For example, we had a dataset with 300+ images we were running an experiment on. With the CPU and only 60 epochs it took about 45 minutes. Using a GPU we could run 200 epochs within a minute. We initially did the experiment using a CPU because we had to troubleshoot CUDA compatibility which is listed in the requirements.txt file which the VENV runs. For those with only CPU capabilities, the performance you get is very limited.

  Solution: Research correct configurations for VENV to allow for optimized performance and apply to scripts.

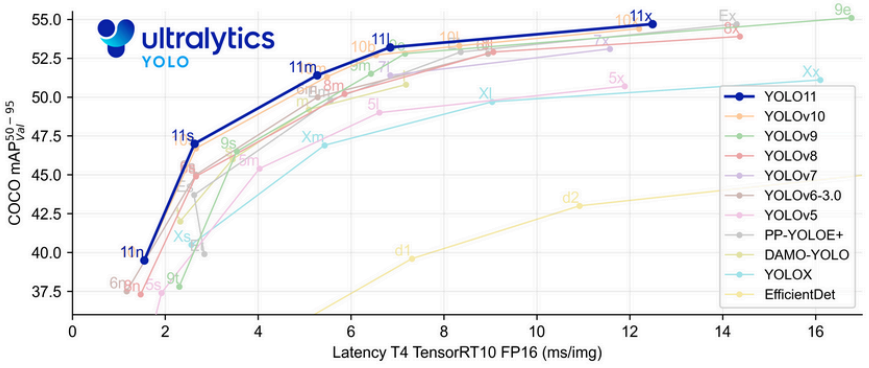
- How can we improve the accuracy of the last project repo?

  Issue: We noticed how the last group annotated their images. They ran a script that placed bounding boxes around the desired plane and then classified the type of plane. This is what's referred to as Object Detection which is great for identifying and giving general idea of where they are in the image, it does not provide detailed information about their exact shape or boundaries.

  Solution: To improve results, we used instance segmentation through a web app called Roboflow . This not only helped identify each plane but also outlined their exact shapes and separated them individually, even if there were multiple planes in one image. This pixel-level accuracy gives the model more detailed information, which can lead to better classification and overall improved performance.

  We also uploaded more data towards the dataset getting over 1000+ images allowing for a variety of data for the model to learn. This led to an increase in accuracy in detecting the engine type for an airplane.

In addition to this we implemented YOLO v11 instead of yolov8 to further improve the accuracy. Yolo v11 builds upon previous versions. In addition to that, we also switched from using YOLOv8 to YOLOv11 to further improve accuracy. YOLOv11 builds on everything the earlier versions did well but brings in newer techniques and optimizations that make it even more powerful. It's faster, more efficient, and better at detecting smaller or more complex objects within an image. By upgrading to YOLOv11, we were able to take advantage of its improved architecture, which helped the model make more confident and accurate predictions. This upgrade played a big role in boosting the overall performance of our system.



- Avoiding Bias when training model

Issue: False negatives due to insufficient data.
Solution: Assign each group member a number of images to scrape and annotate providing as much data as we can to the model. We gathered approx 1000. Images as opposed to the first experiment which only had approx. 300 images.

Non-Technical

- Getting started with last groups project

Issue: None of us have worked with computer vision architectures before and we had to learn how the last repo worked.

Solution: Meetings with professors helped to provide guidance. Mr. Adams provided a video tutorial that taught us how to identify airplanes using YOLOv8 : Image Classification using YOLOv8. From here and a few other related videos we were able to understand how the last project's repo worked and we could begin to implement our solution.

- Gathering data/Annotating data

Issue: How to gather images, how to annotate as a group.

Solution: Looking at the past repo, we noticed they used a bing images web scraper. We've tailored the previous script towards our needs and also included a google web images scraper for additional resources to draw from. Using Roboflow, we could all upload our images in one dataset and assign each other a certain number of images to annotate. We would assign each other 200+ images for the week to get some images into our dataset. Roboflow would then split the pictures into 3 folders(training,validation,testing).

4. Experimental Results

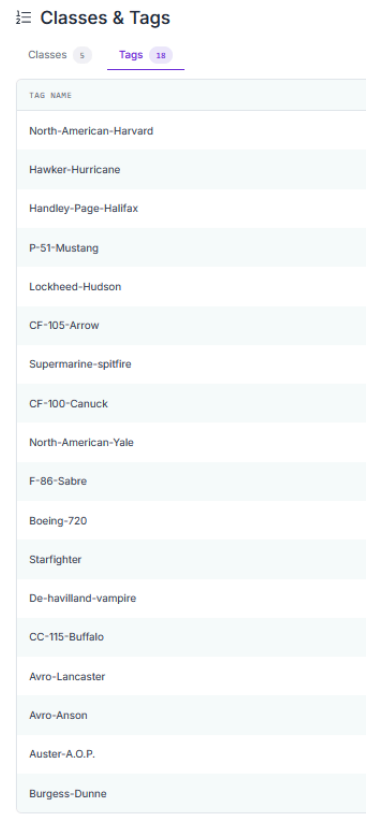
Datasets

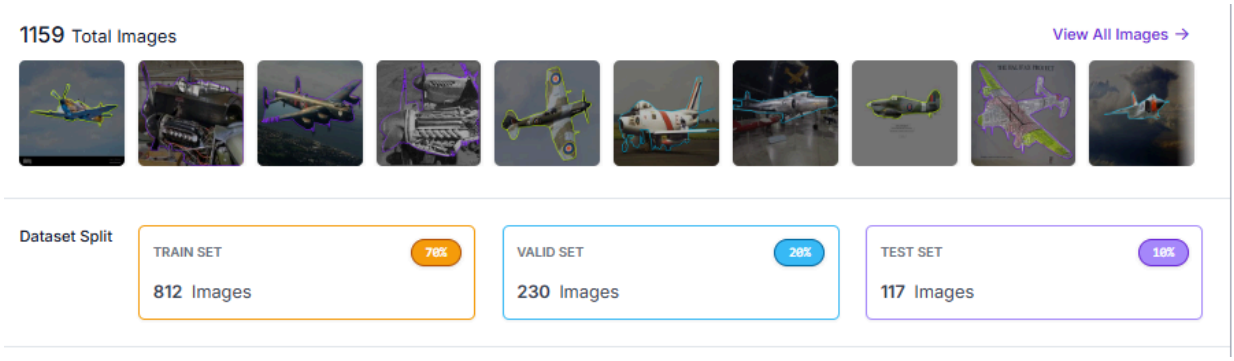
Experiment 1 ~300 images, currently of 4 planes:

- Avro Lancaster 62 images
- Focke-Wulf Fw 190 89 images
- Halifax Bomber 137 images
- P51 Mustang 121 images
- Supermarine Spitfire 61 images

70/20/10 split of training, test, validation;

Experiment 2

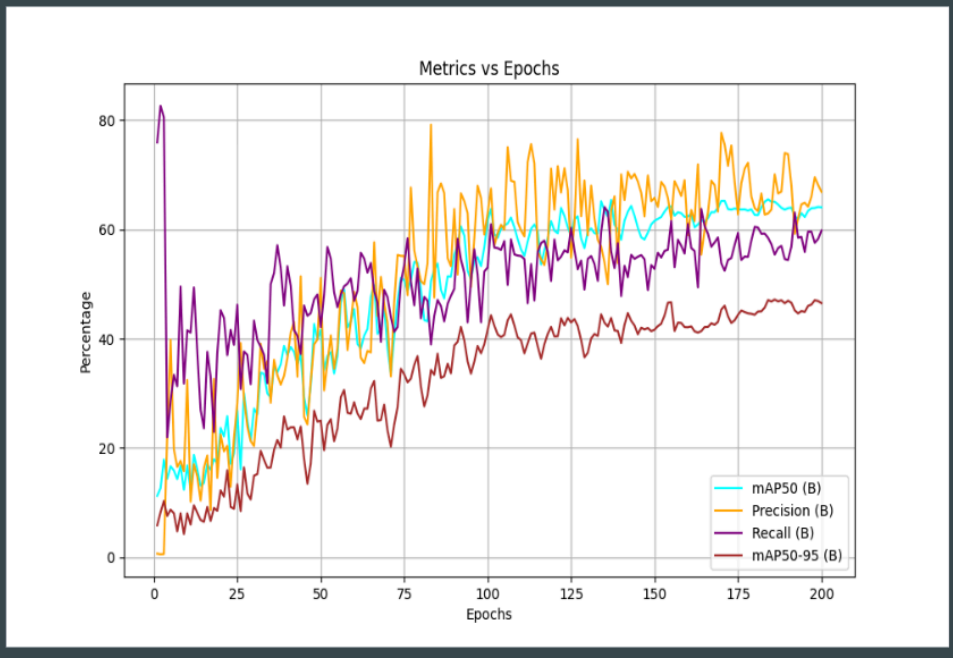




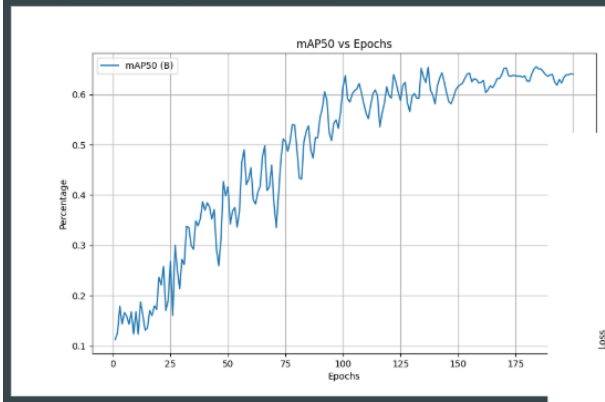
Model Performance

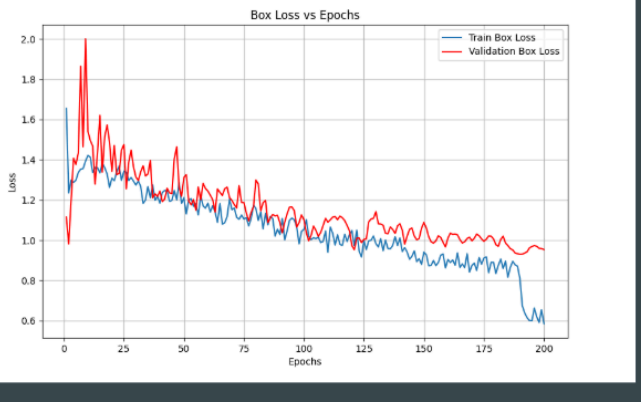
Experiment 1

Metrics

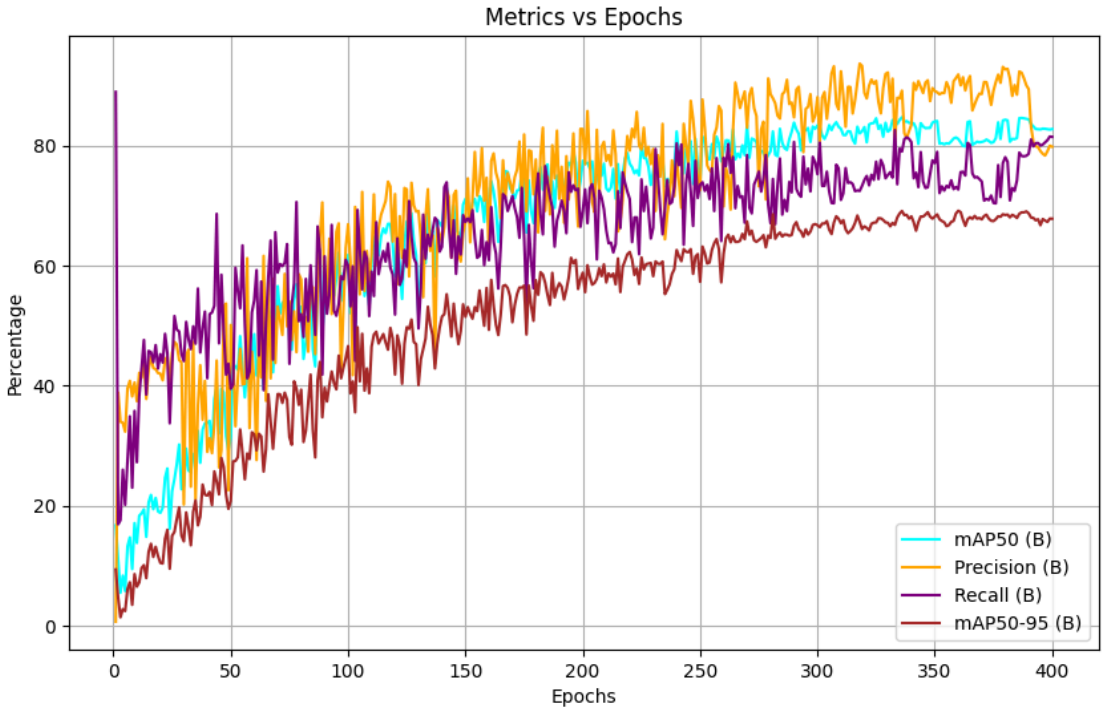


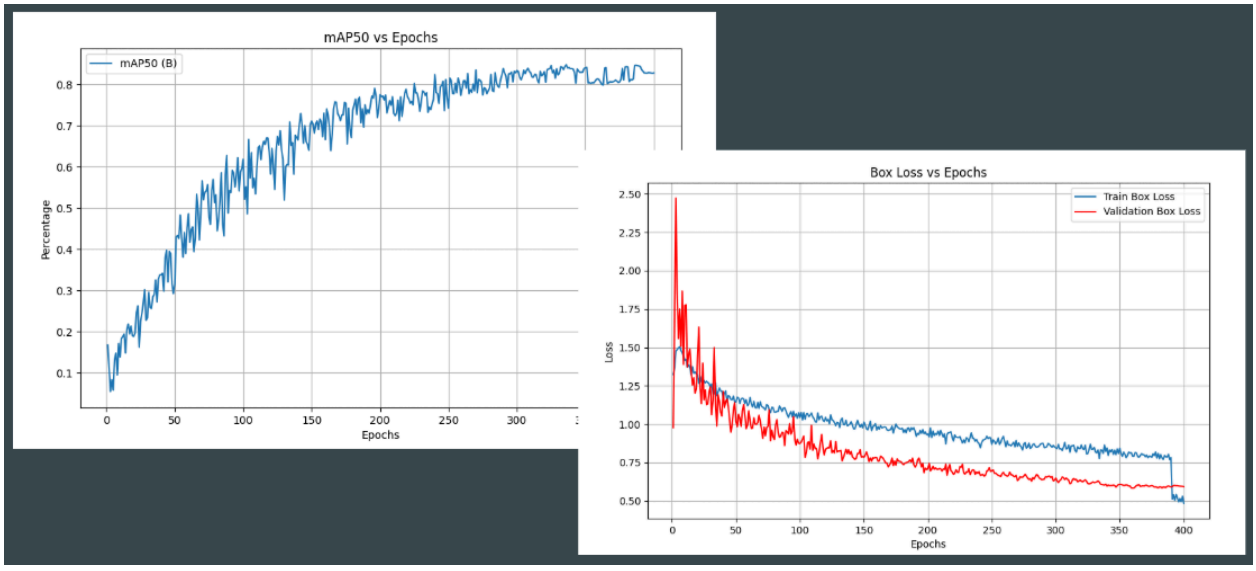
Metrics





Experiment 2





Metrics

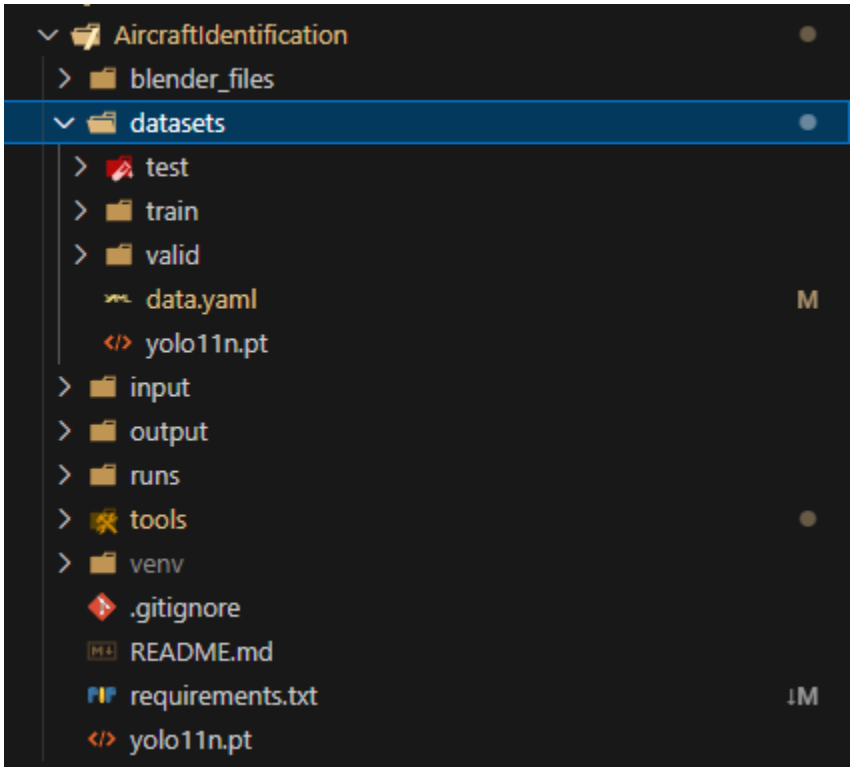
- On both the training and validation
    - Box loss - percentage of how much the model's annotated box is overlapping with the human annotated one
    - Precision - how many correct positive predictions to the total positive predictions; how often it's made a positive correct prediction when a plane was identified
    - Recall - how many true positive predictions to the total positive instances - counteracts precision in the way it'll highlight if the model is simply not identifying anything if it's not

confident. i.e. want recall and precision to be ~equal in our use case as the two correlating means the model isn't simply not identifying specific types of planes and getting it correct based on that.

Going from experiment 1 (first half of the semester) to experiment 2 (second half of the semester), the model had across the board increases in all metrics. mAP50 is up ~20%, validation box loss is down ~50% and precision/recall are both up ~20%. The general metrics are vastly improved and the model is overall much more reliable with detecting and correctly identifying the plane type.

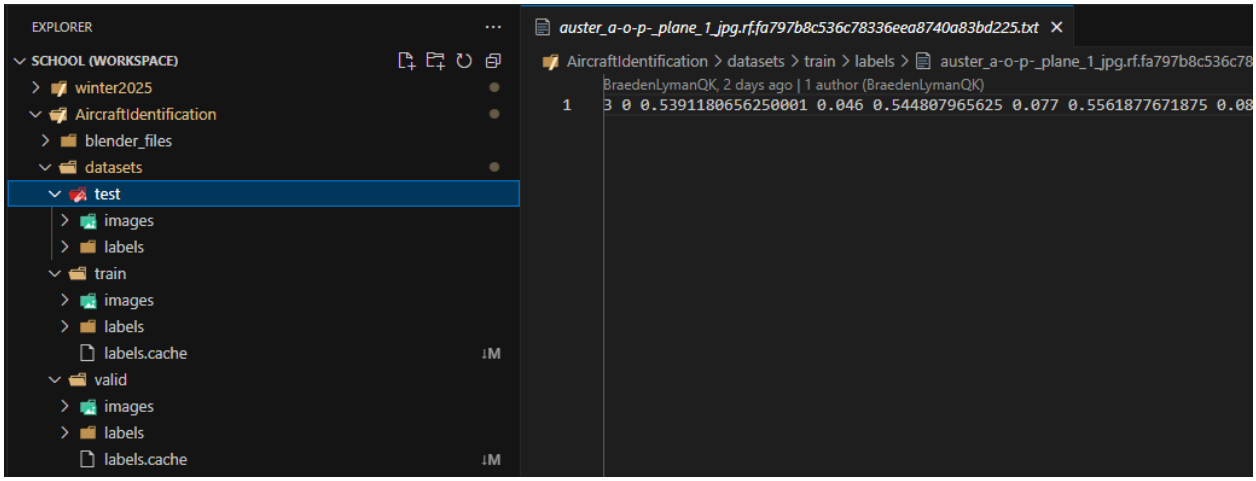
Visually just watching model's annotated videos/images the boxes are more accurately identifying the proper edges of planes, getting the correct classification and in videos much less flickering between either classes or a plane being detected at all.

5. Technical Details

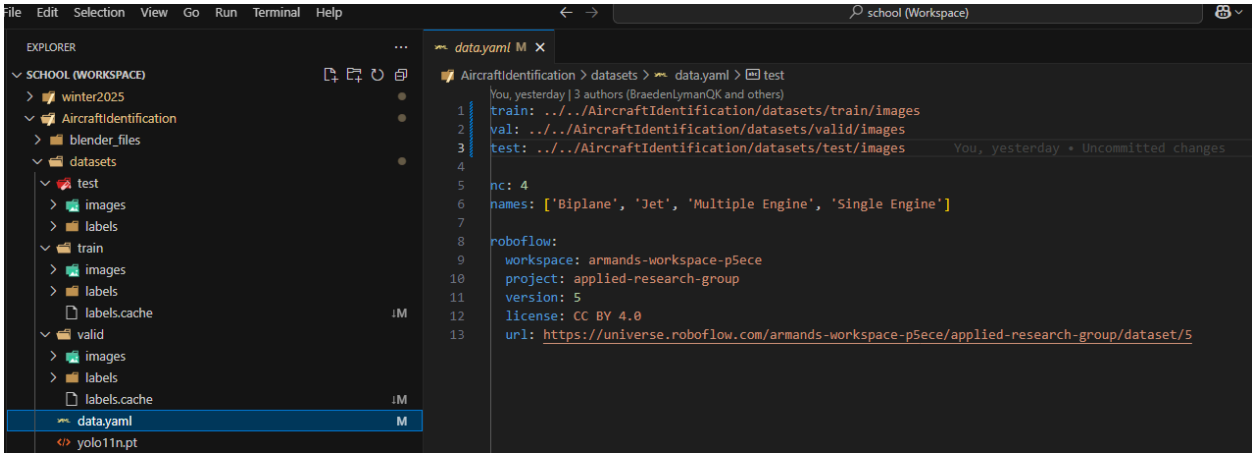


1. To train a model one must provide data. The dataset is divided into 3 subsections: train, valid, and test.
   The training set(train) is the largest portion of the data, and used to "teach" the model by helping it recognize patterns and learn from these examples.
   The validation set(valid) is used to help fine tune the model during training. In this specific folder the model will be fed similar but different images from the training set. The

images may be augmented, transformed, flipped, etc. to help further adjust the parameters of the model.
The testing set(test) contains never before seen images and is used after the model is fully trained. From here we can evaluate its performance and see if the model can accurately detect what we trained it to detect. We used a web application called: http://roboflow.com to compile our dataset and annotate our images. It also handles the splitting of the 3 folders.



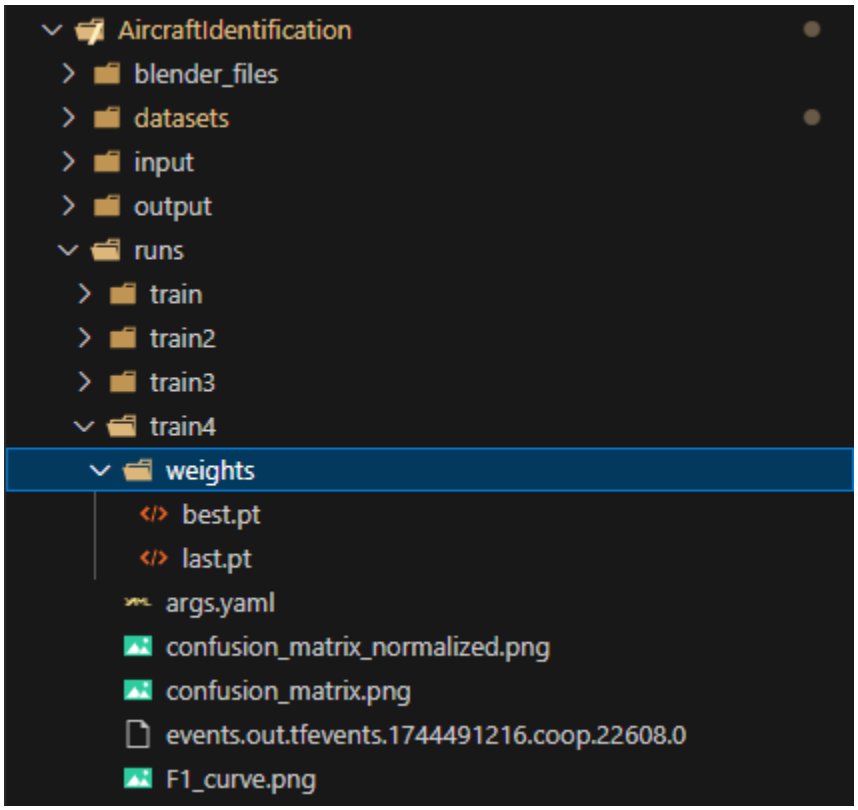
2. Note that each set contains an images and labels folder. With each image comes a corresponding label coordinate. When we open one of the labels as shown in the corresponding image above we can see within each line a single digit number - indicating which classification it is and its corresponding coordinates. We can also see from this label that there have been 3 instances of this particular plane annotated within this particular image.



3. The data.yaml file will contain the instructions on how the model will be trained. You can see the file path towards the datasets, this is how the model knows where to get the images, and how it knows how to classify each plane.

4. With the dataset correctly configured we can begin to train the model. In the corresponding picture in the train_model.py script, you can see the specified model we

used in this experiment and the link to the data.yaml file. The parameters in the train model include, how many epochs to run, augmentations, etc. After specifying the model all you need to do is run model.train() with your specific parameters to train the model. This script is found in the /tools/model_functions path.



5. Once the model is trained the results are stored in a train folder within the runs/train{n} folder.

   The /weights/ folder stores the model's weights, these are how the model knows what classification it's looking at. The YOLO model is a base layer in which the computer image algorithm can pick out shapes and figures, the weights are a layer on top of this that then allow us to classify what we want.

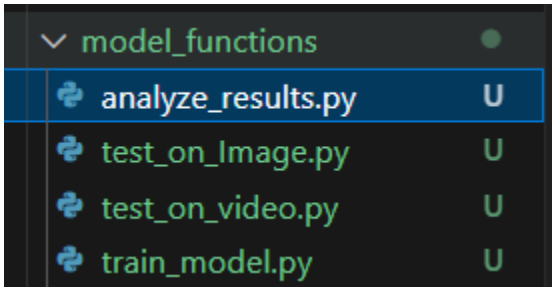
   best.pt – This is the best-performing version of the model during training. It gave the most accurate results on the test data.
   last.pt – This is the final model at the end of training. Sometimes it's slightly less accurate than best.pt, especially if the model started to overfit or struggle.
   We use best.pt when we want the most reliable version of the model to detect objects in real life. You can use either model to test.

   results.csv – Reports how well model performed

This file records how the model performed across different training rounds (called epochs). It logs things like:
How well it detected objects (precision, recall),
How accurate it was at drawing boxes around items,
How fast and efficient it was,a dn ultimately helps us understand whether the model is improving or not.

The other remaining files provide other metrics users can use to further understand the model's results. And the batches of images are showing how the model learns in steps.

6. Under the tools > then model_functions folder you can find:



Analyze_results:

```
"""
This script generates plots from the results.csv file
on a specific training run
Note: results.csv files are under runs/train{n}/results.csv
"""

#Indicate absolute path to results.csv, found in runs folder
results_path = r'C:\school\AircraftIdentification\runs\detect\train27\results.csv'

# Load the results from the CSV file
results = pd.read_csv(results_path)

# Clean the column names by stripping any leading/trailing whitespace
results.columns = results.columns.str.strip()

#1st important one according to tutorial
# Plot train/box_loss and val/box_loss vs epochs
plt.figure(figsize=(10, 6))
plt.plot(results['epoch'], results['train/box_loss'], label='Train Box Loss')
plt.plot(results['epoch'], results['val/box_loss'], label='Validation Box Loss', c='red')
plt.grid()
```

This script requires you to make a path towards the specific train sessions you want to inquire about and then generates a few metrics using matplotlib. We have train/box_loss and val/box_loss vs epochs, maPB50 vs epochs, mAP50 and other metrics vs epochs, being graphed. These three were the most important ones based on the tutorials we watched; however, users are more than welcome to graph other metrics listed in the results.csv folder.

```python
import numpy as np
import cv2
import matplotlib.pyplot as plt
from ultralytics import YOLO

# Load the trained model
model = YOLO(
    r"C:\school\AircraftIdentification\runs\detect\train27\weights\best.pt"
)

# Predict on an image
image_path = r"C:\Users\coops\Downloads\maxresdefault.png"
results = model(image_path)  # Run inference

# Plot results with bounding boxes
annotated_frame = results[0].plot()  # Auto-draws bounding boxes & labels

# Convert from BGR to RGB (Matplotlib uses RGB)
annotated_frame = cv2.cvtColor(annotated_frame, cv2.COLOR_BGR2RGB)
```

The test_on_image.py script requires you to specify which model you would like to use, and if you can recall the models you trained are in the runs/detect folder. Once selected you can just specify an image you would like to test in the image_path variable and hit run.

```python
#Load your video you want to test
sv.process_video(
    source_path=r"C:\Users\coops\Downloads\p51mustangvid1.mp4",
    target_path=r"C:\school\AircraftIdentification\output\output3.mp4", # where you want to save the video
    callback=callback
)
```

The test_on_video.pyscript works similarly to the image script. Specify the model you want to test and the path of your model, and the only difference here is you must specify a video(mp4) and the target path of where you would like to save the video.

GitHub Repository Structure

```
├── blender_files
├── datasets
│   └── Train, Test, Valid
├── input
│   └── input folder for any images/videos to annotate
├── output
│   └── output folder for any images/videos to annotate
├── runs
│   └── output folder for train weights and + measure values
├── tools
│   └── tools to run and train the project
├── venv
```

```
│   └── virtual environment directory - created during setup
├── .gitignore
├── README.md
├── requirements.txt
└── yolo11n.pt
```

6. Conclusion & Future Work

Achievements

- Annotated 1000+ images
- Updated Github repo (clean)
- Added in virtual environment
- mAP50 is up ~20%, validation box loss is down ~50% and precision/recall are both up ~20%.

Future Enhancements

- More blender models pictures (take pictures from various angles)
- More data the better
- More training/testing
- Strip text from planes and compare to the CASPIR dataset for further validation
- Include wireframe / higher resolution segmentation

Information for Future Groups

- Using roboflow help drastically with annotating our training data
- Using yolo11 helped improve accuracy and results from past groups.
- Ensure the data is clean, make sure the class and label for that plane is precisely for that plane. Obtaining and cleaning the data is the most important part
- Use the tools provided in the model functions folder. We've made it so you can test the model via images and videos.
- Using tutorials gave us a great base to learn the previous groups repository: IMAGE CLASSIFICATION with Yolov8 custom dataset | Computer vision tutorial
- Split annotations amongst your group. By assigning weekly images for your group to complete it makes it more manageable and you can increase the size of your dataset gradually.

Don't reinvent the wheel, there's open source or at least free tools for almost everything, take 10-15 minutes to see what's out there before committing to making your own tool. In our case Roboflow for annotating data saves a lot of time. It's got a magic pen selector that does the bulk of annotating data for us. In general, taking the time to understand the tools and the

structure of stuff, debugging, adding features and modifications will be significantly easier if you understand the underlying concepts; especially YOLO and how projects need to be structured.

Split annotations similar to how many of the official YOLO datasets are set up would be a great starting point for either reannotating our data or new data. You can get much finer control over what stuff looks like and as a result the model is more accurate overall.

Document what you've used and any problems you run into, if it's a problem for you, it'll probably be a problem for someone else. Our installation guide and how the repo is structured is a result of this, we tried to keep it simple and with as little bloat as possible. Problems are inevitable, figure out what is actually a problem and find a solution if feasible. Patch solutions are usually permanent once the program runs (Our direct file paths for most of the more frequently changed options are a consequence of this).