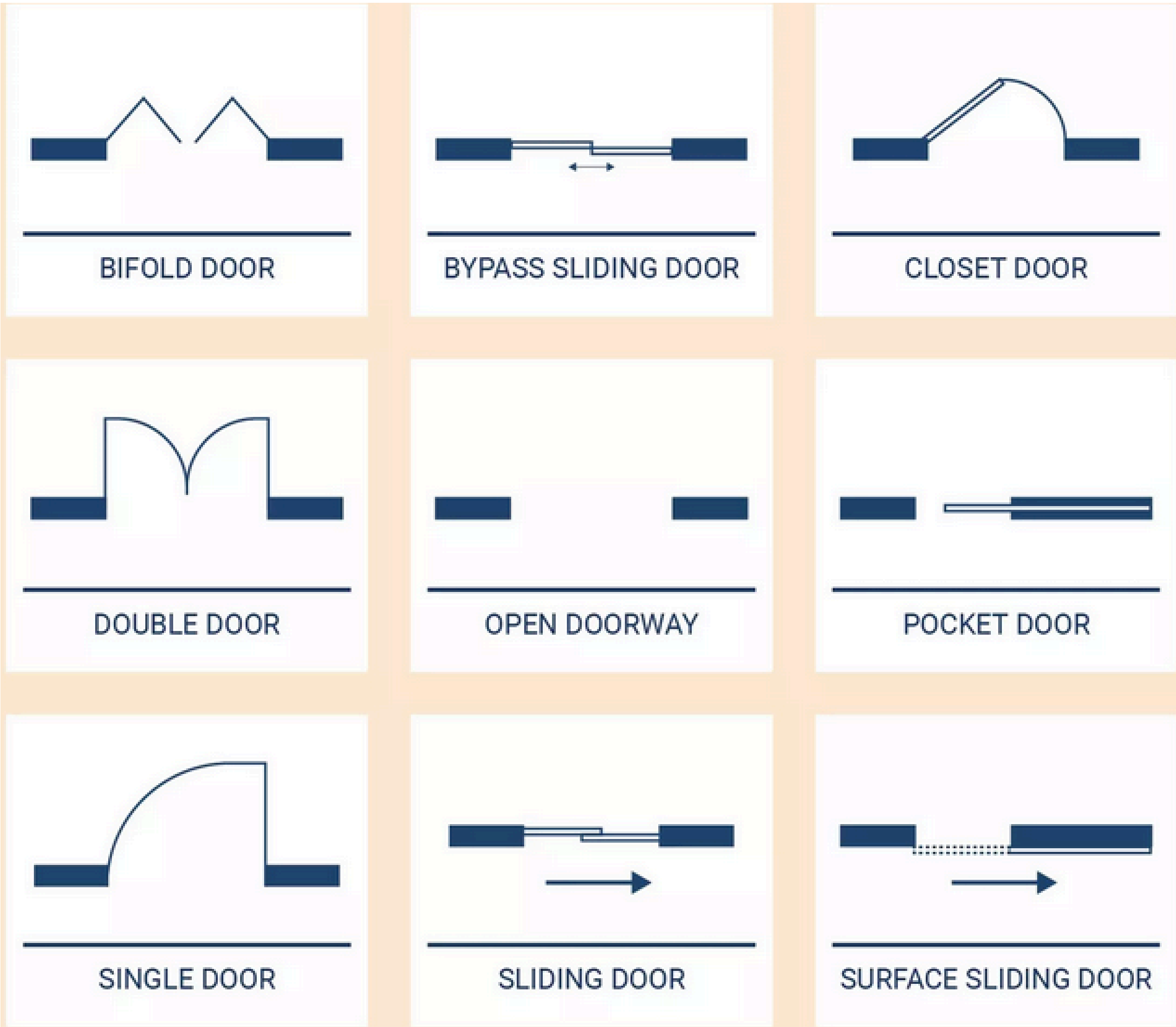
doors-and-windows

Step 0: Preliminary

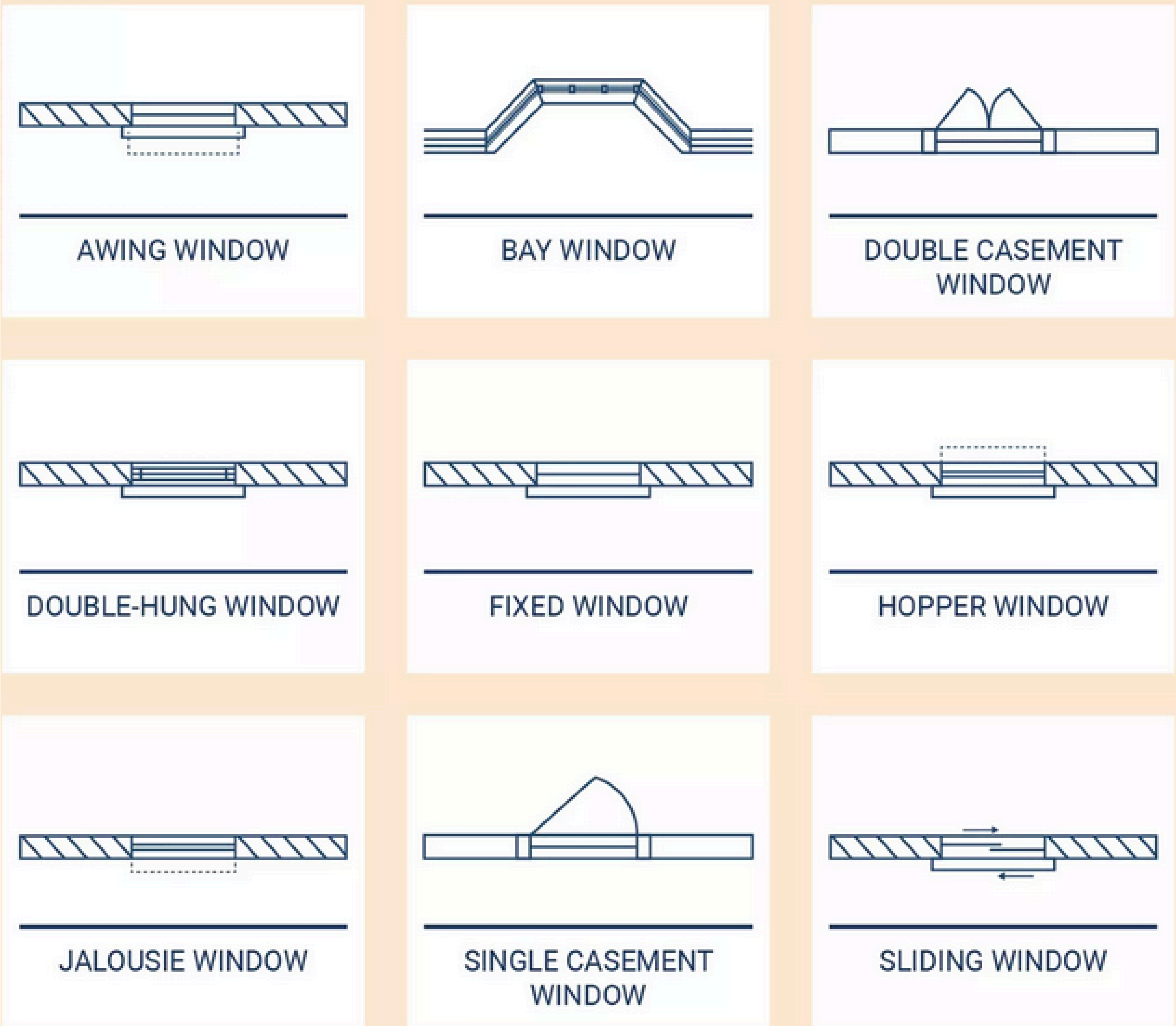
The main challenge behind `doors-and-windows` lied behind the first step: labelling. To properly execute this assignment, one had to first learn how to read architecture plans, and understand the subtle differences between each symbols in order to distinguish floors, walls, windows, etc.

The scope of the task was to identify the doors and windows of a floor plan. Upon searching in the internet, there were many references available into the floor plan symbols and abbreviations.

The doors of a floor plan are identified like so:



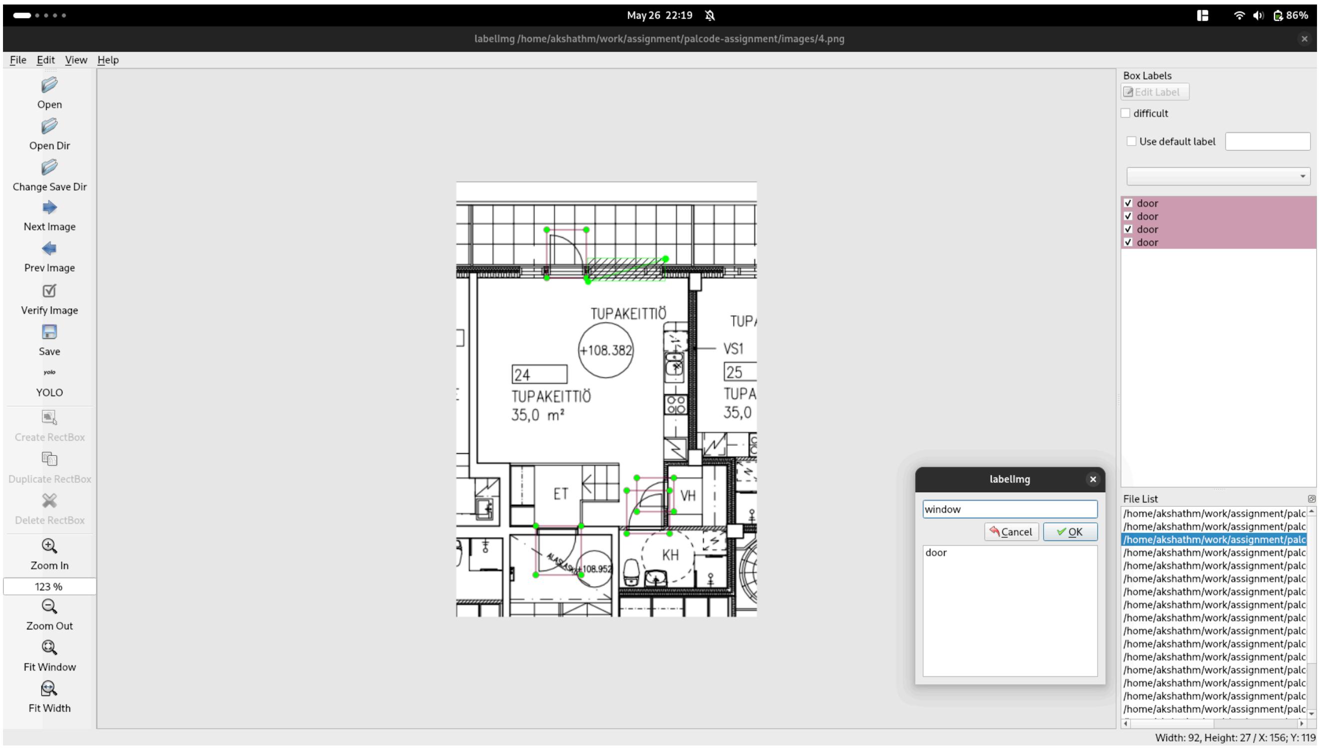
While the windows of a floor plan are represented like so:

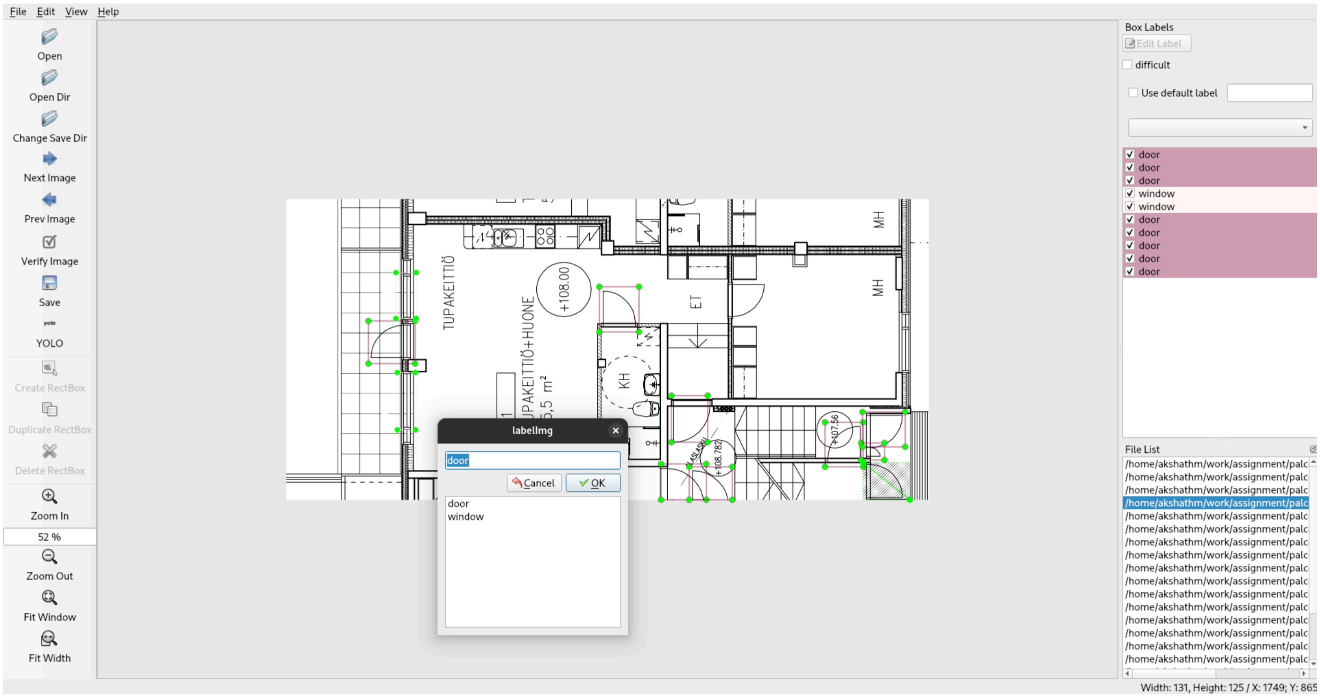


Step 1: LabelImg

`labelImg` is a well-known Python library that allows you to label images with bounding boxes in order to later, train a machine learning model. Our first step involved analysing the dataset that had been given to us, from which there were 22 images in total, 7 of which, were duplicates.

As part of the pre-processing step before labelling, we had to remove the duplicates in order to prevent the model from "memorising" the locality of labels. Next step, involved labelling itself.





Step 2: Setting up the dataset

Given the validation split to be 0.2, our 15 images will be split into 12 training and 3 validation images, and our dataset to be structured like so:

```
├── images/
│   ├── train/
│   └── val/
├── labels/
│   ├── train/
│   └── val/
```

```
├── classes.txt
├── data.yaml
```

with our `data.yaml` containing the following structure:

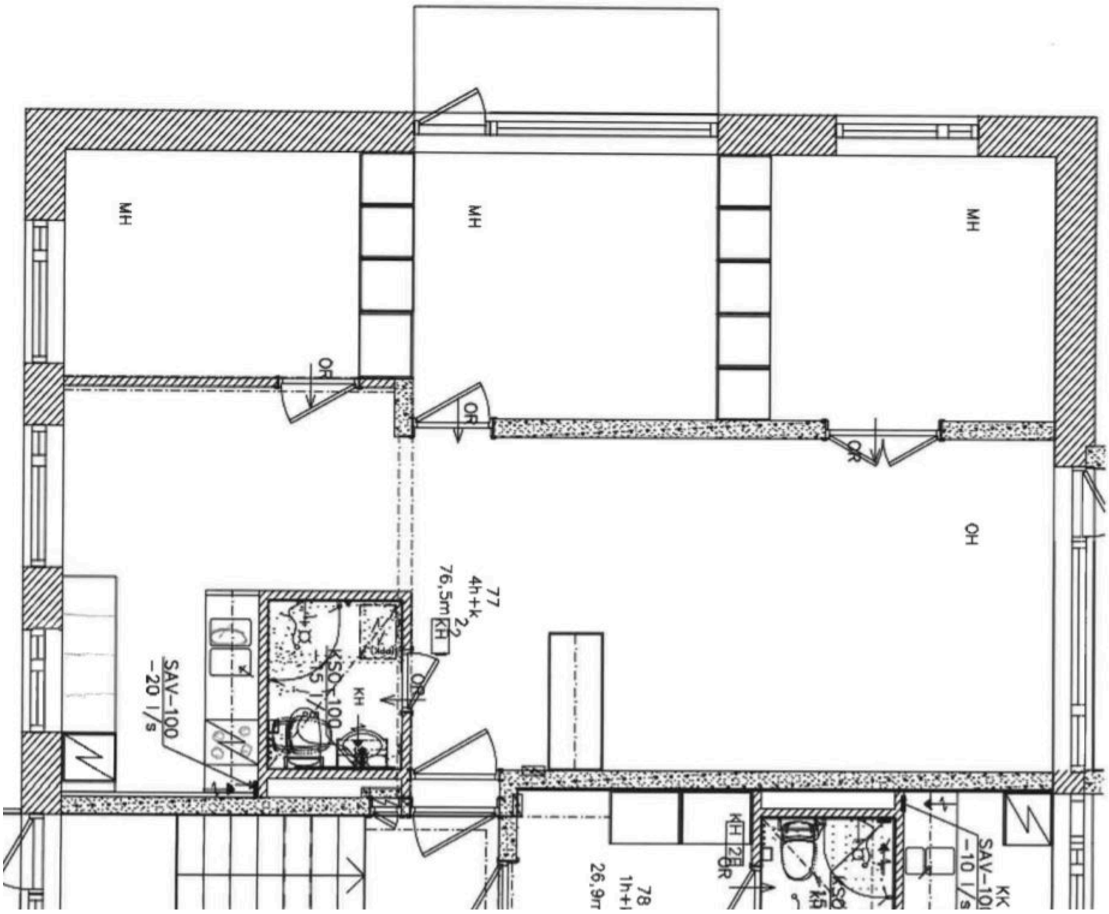
```yaml
train: ./images  # same for val if you're just experimenting
val: ./images

nc: 2
names: ['door', 'window']
```

and our `classes.txt` containing the number of classes for our use-case. In this case, it's two:

```
door
window
```

Our example image and our annotated .txt file will look like so:



14.txt (same name as image)

```
1 0.027879 0.327676 0.031993 0.159269
1 0.029250 0.552219 0.034735 0.151436
1 0.029707 0.750000 0.037477 0.113577
0 0.238574 0.436031 0.063985 0.070496
```

```
0 0.345978 0.454961 0.063071 0.066580
0 0.343236 0.912533 0.066728 0.060052
0 0.348720 0.830287 0.072212 0.067885
0 0.670475 0.494778 0.085009 0.057441
1 0.685558 0.157963 0.106033 0.049608
0 0.341865 0.133812 0.058501 0.069191
1 0.455210 0.152089 0.175503 0.037859
1 0.820384 0.723890 0.021024 0.259791
```

where 0 is `door` and 1 is `window`.

Step 3: Roboflow + YOLO

For ease of use, the dataset was published to Roboflow as a move to streamline the process. Since Roboflow is easily integrated and usable with YOLO, the dataset was first published to Roboflow and three YOLO variants were trained:

1. yolov8n :- nano variant
2. yolov8s :- small variant
3. yolov8m :- medium variant

Why three models?

Because of hardware constraints, `yolov8n` was the first-choice in consideration. That is, until the model results were actually interpreted. The main goal became to assess the model variants and identify the optimal model for the task given.

Validation Metrics:

Model	Params (M)	GFLOPs	Inference Speed (ms/image)	mAP@50	mAP@50-95
YOLOv8n	3.01	8.1	32.65	0.231	0.119
YOLOv8s	11.13	28.4	74.78	0.411	0.239
YOLOv8m	25.84	78.7	158.06	0.406	0.235

Class-wise Performance Metrics:

Class-wise Performance (Precision and Recall)

Model	Class	Precision	Recall
YOLOv8n	0	0.945	0.206
	1	1.000	0.000
YOLOv8s	0	0.682	0.559
	1	0.290	0.267
YOLOv8m	0	0.807	0.559
	1	0.165	0.433

From this analysis, `yolov8n` was the fastest and lightest model however it severely under-performed in terms of recall, especially for Class 1 (windows), where recall was 0. This makes it impractical for use despite its speed.

`yolov8s` provided the best balance in terms of accuracy, recall and inference time. It achieved the highest mAP@50 and mAP@95 across all models. Both classes showed reasonable recall despite windows being a much more difficult class to predict, compared to doors.

`yolov8m` had slight lower mAP scores compared to `yolov8s`, despite the model being significantly slower and heavier. The performance did not justify the increase resource consumption, making it a suboptimal choice, along with `yolov8n`, given the hardware and deployment constraints.

But why did these models perform poorly in the first place?

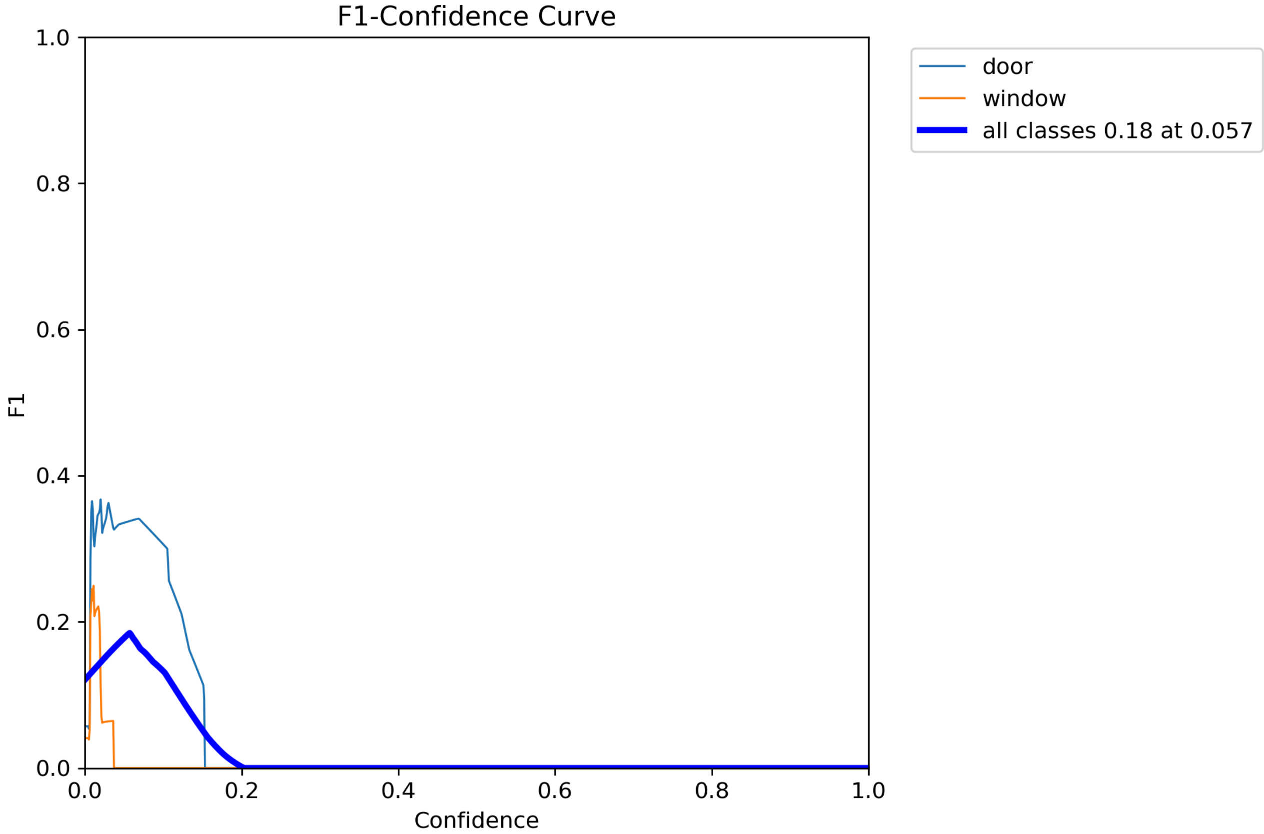
There are three very critical parts to a model training life-cycle. They are:

- dataset preparation
- model selection
- hyper-parameter tuning

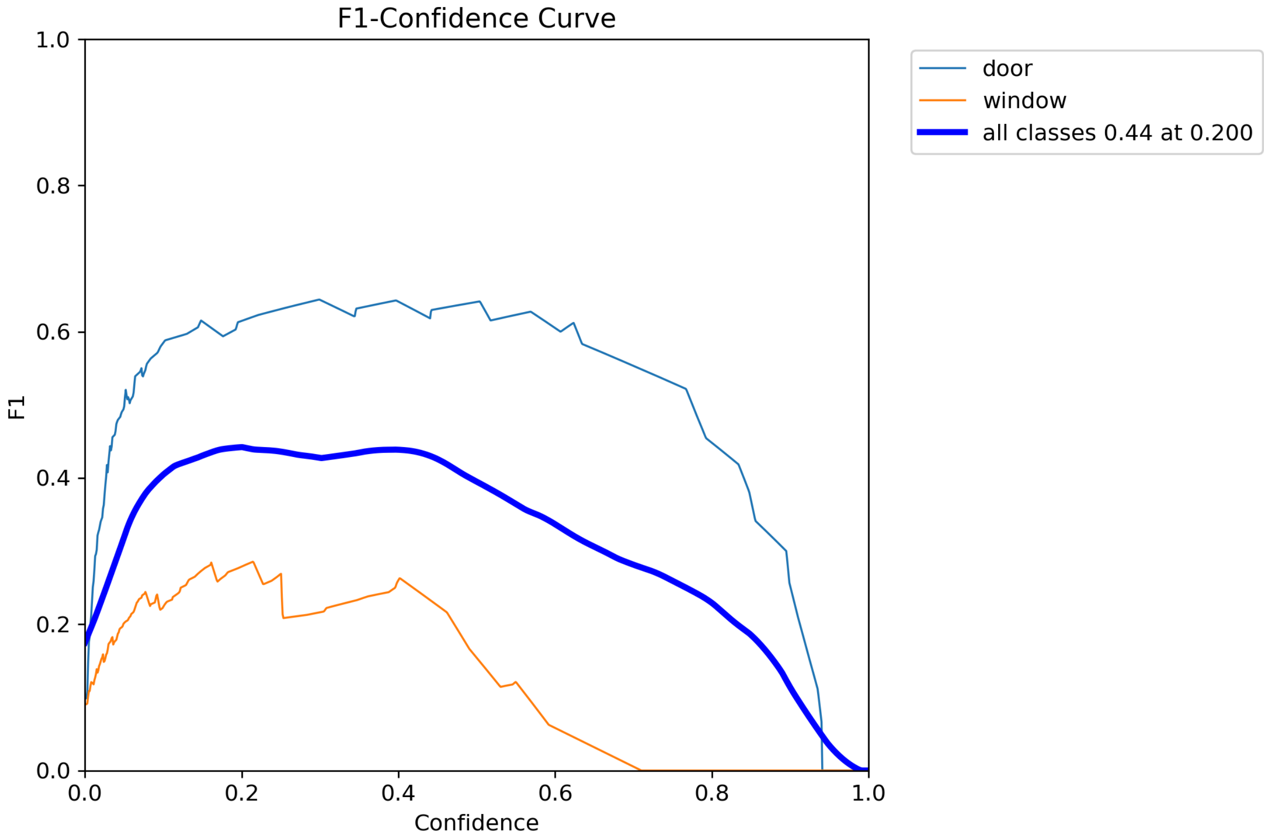
Any one of these aspects were to be sub-par, the task cannot be trained properly, nor can the end-user or a research interpret any valuable insight from it. Such was the case of this task.

The main challenge lied in the shortage of size of dataset. With just 12 training images, which were increased to 38 from data augmentation, and 3 validation samples, it is difficult for any model to learn any suitable insight into the nature of the input. As a consequence, model selection had to be done more carefully and `yolov8s` was the winner of the experiment.

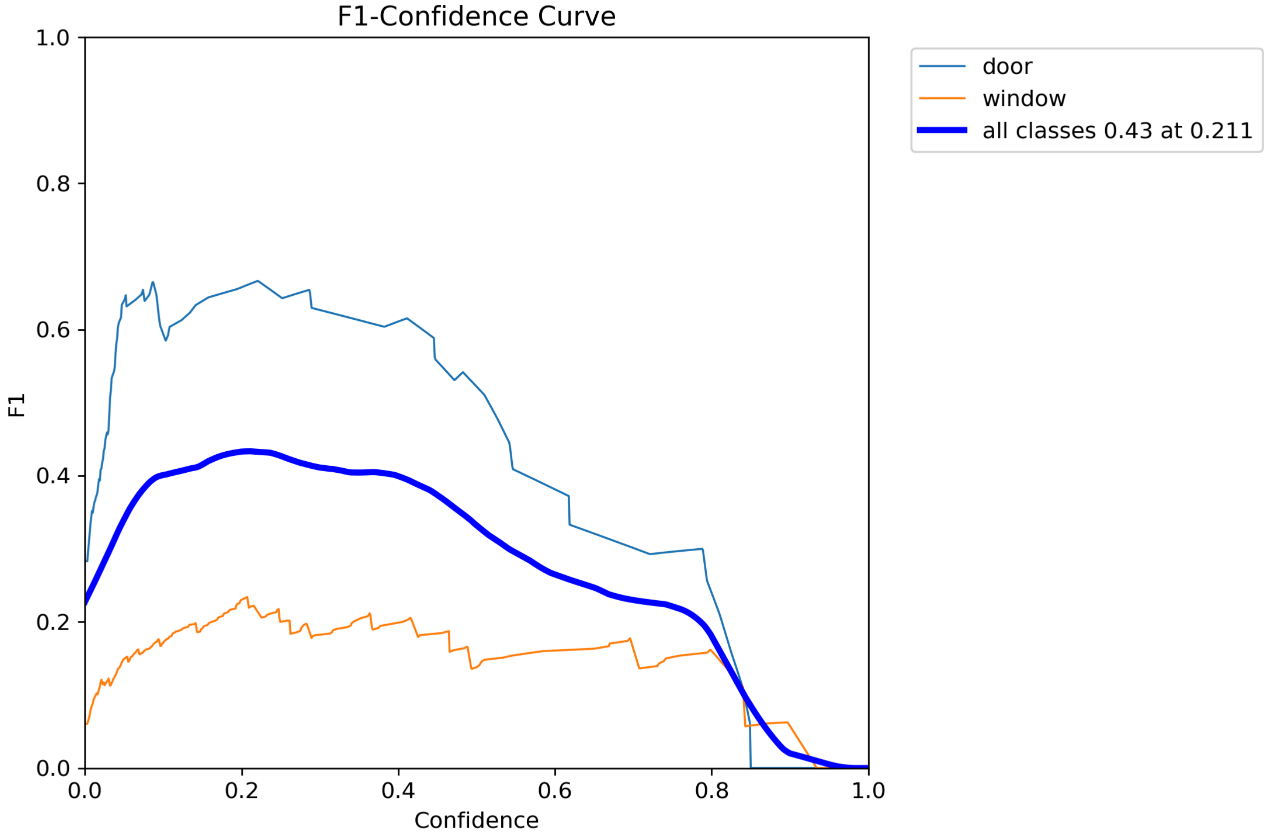
This is the **F1-curve** for `yolov8n`



And for `yolov8s`



And for `yolov8m`

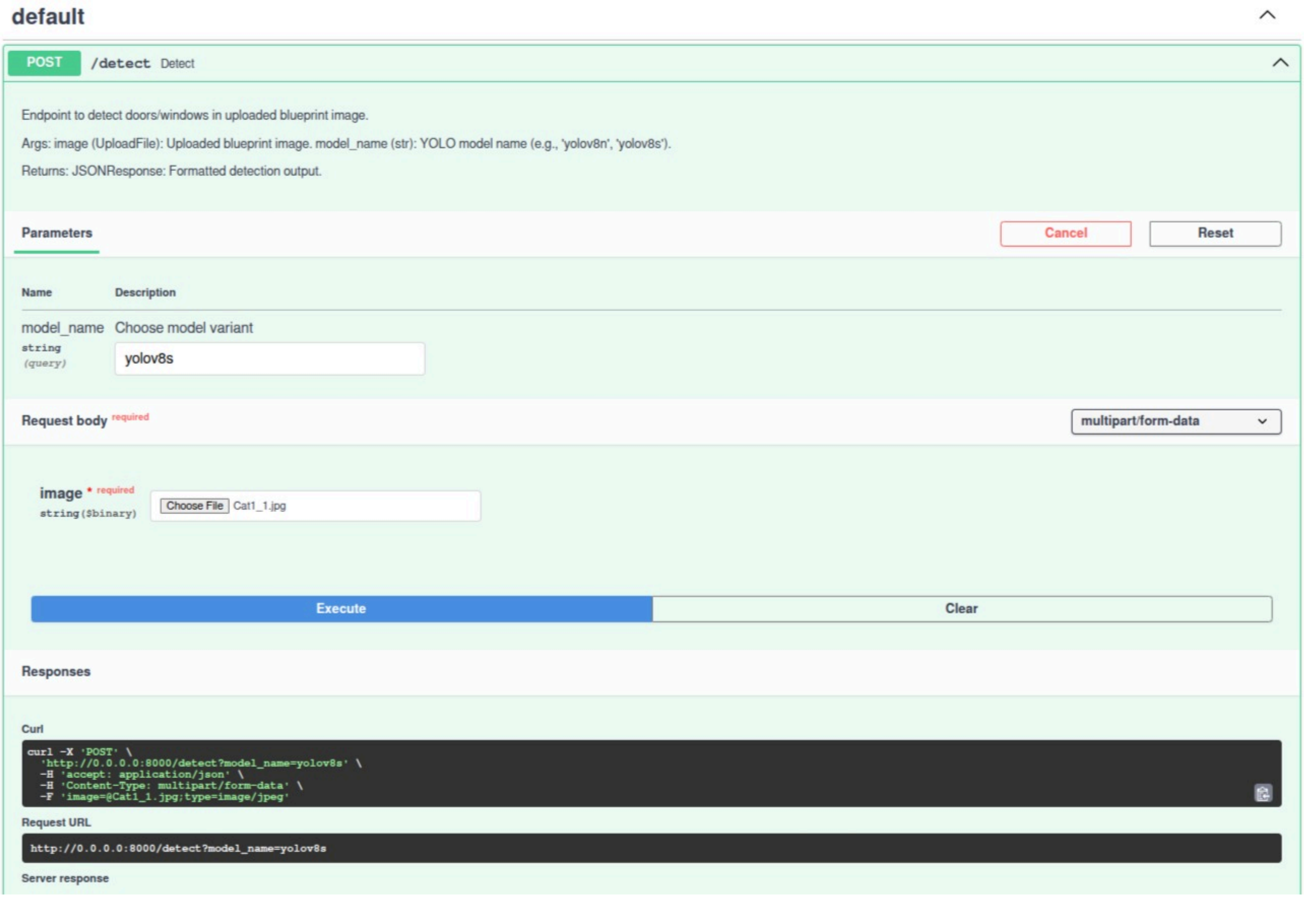


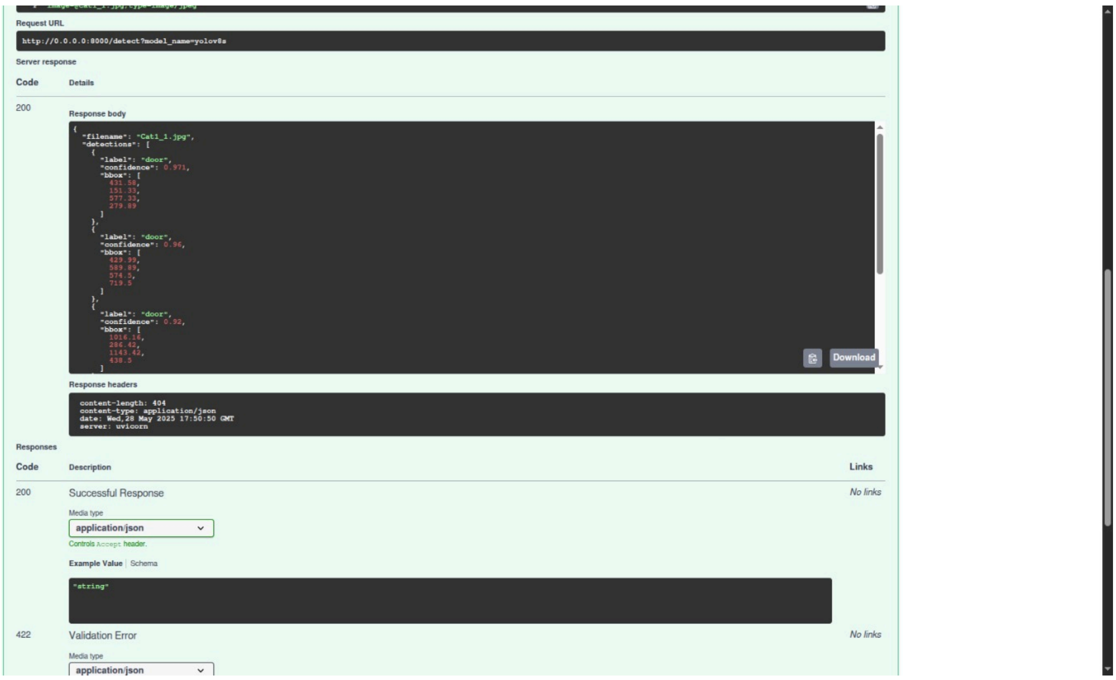
Step 4: API Development

The initial setup of the application was a `FastAPI` backend that called the model upon the user's request and returned the bounding boxes or the JSON structure as described in the README.md

This however, was not enough. The final version of the application became a Streamlit front-end, which included a minimal UI, as well as visualization of the locations labeled by the model.

Interface:





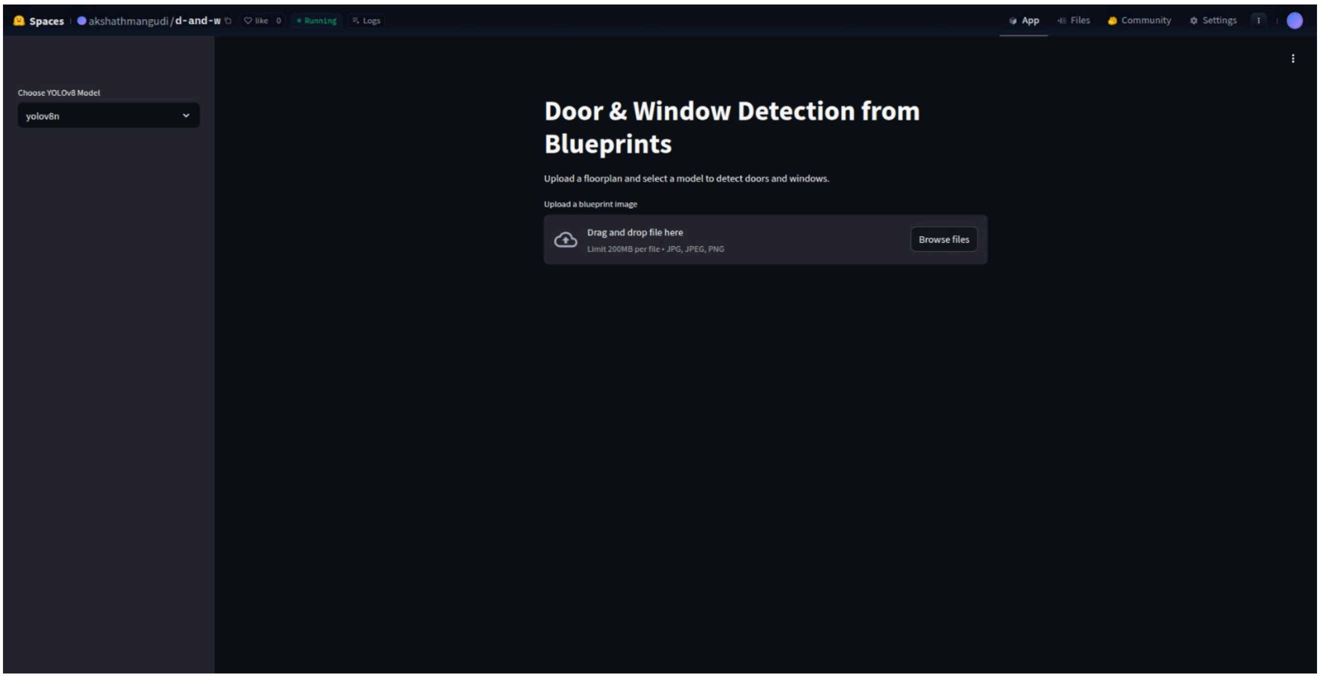
Step 5: Deployment

The application has been deployed to:

- HuggingFace Spaces
- Github Container Registry

Both the `streamlit` and `docker pull` versions are available for use. To see more details on installation, kindly refer to the README.md

Interface:



Spaces
akshathmangudi / d-and-w
like 0
Running
Logs
App
Files
Community
Settings
Choose YOLOv8 Model
yolov8n
Door & Window Detection from Blueprints
Upload a floorplan and select a model to detect doors and windows.
Upload a blueprint image
Drag and drop file here
Limit 200MB per file • JPG, JPEG, PNG
Browse files