**TASK1:Grain size analysis in python using a microscope image:**

* Understanding grains in metal alloys is important for quality control and research.Optical and electron microscopes are used to image well-polished alloy surfaces to capture grain structure. Automating the process of grain analysis not only speeds up QC/research but also yields consistent & repeatable results. It performs various image processing and analysis tasks using libraries such as OpenCV, NumPy, Matplotlib, SciPy, and scikit-image and preprocessing tasks are being done in google colab notebook.
* There were two ways to process the particles:

1.using thresholding and converting them into different colours using opencv

2. watershed algorithm using opencv

.**1. thresholding and converting them into different colours using opencv**

* Various operations were used in this method and those method were:

1.Denoising

2,grayscaling

3.thresholding

4.erosion and dilation

5.labeling

6.color to labels

Basic pseudo algorithm for method are:

**Step 1:** Read image and define pixel size (if needed to convert results into microns, not pixels)

**Step 2:** Denoising, if required and threshold image to separate grains from boundaries.

**Step 3:** Clean up image, if needed (erode, etc.) and create a mask for grains

**Step 4:** Label grains in the masked image

**Step 5:** Measure the properties of each grain (object)

**Step 6:** Output results into a csv file

Code:

import cv2

import numpy as np

from matplotlib import pyplot as plt

from scipy import ndimage

from skimage import measure, color, io

**#STEP1 - Read image and define pixel size**

img = cv2.imread("images/grains2.jpg", 0)

pixels\_to\_um = 0.5 # (1 px = 500 nm)

**#Step 2: Denoising, if required and threshold image**

**#No need for any denoising or smoothing as the image looks good.**

**#Change the grey image to binary by thresholding.**

ret, thresh = cv2.threshold(img, 0, 255, cv2.THRESH\_BINARY+cv2.THRESH\_OTSU)

**#Step 3: Clean up image, if needed (erode, etc.) and create a mask for grains**

kernel = np.ones((3,3),np.uint8)

eroded = cv2.erode(thresh,kernel,iterations = 1)

dilated = cv2.dilate(eroded,kernel,iterations = 1)

**# Now, we need to apply threshold, meaning convert uint8 image to boolean.**

mask = dilated == 255

io.imshow(mask) **#cv2.imshow() not working on boolean arrays so using io**

**#Step 4: Label grains in the masked image**

**#Now we have well separated grains and background. Each grain is like an object.The scipy ndimage package has a function 'label' that will number each object with a unique ID.The 'structure' parameter defines the connectivity for the labeling. This specifies when to consider a pixel to be connected to another nearby pixel, i.e. to be part of the same object.**

**#use 8-connectivity, diagonal pixels will be included as part of a structure.this is Image default but we have to specify this for Python, or 4-connectivity will be used 4 connectivity would be [[0,1,0],[1,1,1],[0,1,0]]**

s = [[1,1,1],[1,1,1],[1,1,1]]

labeled\_mask, num\_labels = ndimage.label(mask, structure=s)

**#color the labels to see the effect**

img2 = color.label2rgb(labeled\_mask, bg\_label=0)

cv2.imshow('Colored Grains', img2)

cv2.waitKey(0)

**#View just by making mask=threshold and also mask = dilation (after morph operations)Some grains are well separated after morph operations.Now each object had a unique number in the image.**

**#Step 5: Measure the properties of each grain (object) regionprops function in skimage measure module calculates useful parameters for each object.**

clusters = measure.regionprops(labeled\_mask, img)

**#Can print various parameters for all objects for prop in clusters:**

print('Label: {} Area: {}'.format(prop.label, prop.area))

**#Step 6: Output results into a csv file**

**Best way is to output all properties to a csv file**

propList = ['Area',

'equivalent\_diameter',

'orientation',

'MajorAxisLength',

'MinorAxisLength',

'Perimeter',

'MinIntensity',

'MeanIntensity',

'MaxIntensity']

output\_file = open('image\_measurements.csv', 'w')

output\_file.write(',' + ",".join(propList) + '\n')

for cluster\_props in clusters:

output\_file.write(str(cluster\_props['Label']))

for i,prop in enumerate(propList):

if(prop == 'Area'):

to\_print = cluster\_props[prop]\*pixels\_to\_um\*\*2  **#Convert pixel square to um square**

elif(prop == 'orientation'):

to\_print = cluster\_props[prop]\*57.2958  **#Convert to degrees from radians**

elif(prop.find('Intensity') < 0): **# Any prop without Intensity in its name**

to\_print = cluster\_props[prop]\*pixels\_to\_um

else:

to\_print = cluster\_props[prop]

output\_file.write(',' + str(to\_print))

output\_file.write('\n')

output\_file.close() **#Closes the file**

**Note:particles are not separated using such a method giving false counting hence we use watershed algorithm to separate the particles.**

**2. watershed algorithm using opencv**

Threshold based segmentation will not yield good results if the features of interest cannot be easily distinguished using the histogram of pixel values. For example, grains in a microscope image (or cells) will not be efficiently separated thus resulting in wrong insights about the sample.

Watershed assisted segmentation is ideal for these situations. The image can be thresholded first using a traditional approach to identify definitely positive and definitely negative regions. Then, watershed algorithms can be used to fill the gaps. This demonstrates the use of watershed algorithm using a microscope image showing grains and boundaries

* Various operations were used in this method and those method were:

1.Denoising

2,grayscaling

3.thresholding

4.Morphological operations(opening)

5.dilation

6.distance transformation

7.labeling

8.color to labels

pseudo algorithm for method are:

* Read the Image:

1. Load an image of grains ("grains2.jpg").

2. Convert the image to grayscale.

* Thresholding and Morphological Operations:

1.Apply Otsu's thresholding to convert the grayscale image to a binary image.

2.Perform morphological opening to remove small noise and clear the image border.

* Identify Sure Background and Foreground:

1.Dilate the opening to identify the sure background area.

2.Use distance transform and thresholding to identify the sure foreground area.

3.Determine the unknown ambiguous region by subtracting the sure foreground from the sure background.

* Marker Creation for Watershed Algorithm:

1.Label the sure foreground regions using Connected Components.

2..Adjust labels to ensure sure background is not labeled as 0.

3.Mark the unknown region with 0.

* Watershed Transformation:

1.Apply the watershed algorithm to fill regions based on the markers.

2.Boundaries are marked as -1.

3.Visualize and Output Results:

* Overlay boundaries on the original image in yellow.

1.Color code the regions based on the watershed segmentation.

2.Extract properties of detected grains using regionprops function.

3.Write the measured properties of each grain to a CSV file

Code:

import cv2

import numpy as np

from matplotlib import pyplot as plt

from scipy import ndimage

from skimage import measure, color, io

img1 = cv2.imread("images/grains2.jpg")

img = cv2.cvtColor(img1,cv2.COLOR\_BGR2GRAY)

pixels\_to\_um = 0.5 **# 1 pixel = 500 nm**

**#Threshold image to binary using OTSU. ALl thresholded pixels will be set to 255**

ret1, thresh = cv2.threshold(img, 0, 255, cv2.THRESH\_BINARY+cv2.THRESH\_OTSU)

**# Morphological operations to remove small noise - opening**

**#To remove holes we can use closing**

kernel = np.ones((3,3),np.uint8)

opening = cv2.morphologyEx(thresh,cv2.MORPH\_OPEN,kernel, iterations = 2)

from skimage.segmentation import clear\_border

**#opening = clear\_border(opening) Remove edge touching grains**

**#Check the total regions found before and after applying this.**

**#Now we know that the regions at the center of cells is for sure cells.The region far away is background.We need to extract sure regions. For that we can use erode. But we have cells touching, so erode alone will not work. To separate touching objects, the best approach would be distance transform and then thresholding.**

**# let us start by identifying sure background area**

sure\_bg = cv2.dilate(opening,kernel,iterations=2)

**# Finding sure foreground area using distance transform and thresholding intensities of the points inside the foreground regions are changed to**

**distance their respective distances from the closest 0 value (boundary).**

dist\_transform = cv2.distanceTransform(opening,cv2.DIST\_L2,3)

**#Let us threshold the dist transform by 20% its max value.**

ret2, sure\_fg = cv2.threshold(dist\_transform,0.2\*dist\_transform.max(),255,0)

**# Unknown ambiguous region is nothing but background - foreground**

sure\_fg = np.uint8(sure\_fg)

unknown = cv2.subtract(sure\_bg,sure\_fg)

**#Now we create a marker and label the regions inside. For sure regions, both foreground and background will be labelled with positive numbers. Unknown regions will be labelled 0. For markers let us use ConnectedComponents.**

ret3, markers = cv2.connectedComponents(sure\_fg)

**#One problem rightnow is that the entire background pixels is given value 0.This means watershed considers this region as unknown.So let us add 10 to all labels so that sure background is not 0, but 10**

markers = markers+10

**# Now, mark the region of unknown with zero**

markers[unknown==255] = 0

**#Now we are ready for watershed filling.**

markers = cv2.watershed(img1,markers)

**#Let us color boundaries in yellow. OpenCv assigns boundaries to -1 after watershed.**

img1[markers == -1] = [0,255,255]

img2 = color.label2rgb(markers, bg\_label=0)

cv2.imshow('Overlay on original image', img1)

cv2.imshow('Colored Grains', img2)

cv2.waitKey(0)

for prop in regions:

print('Label: {} Area: {}'.format(prop.label, prop.area))

propList = ['Area',

'equivalent\_diameter', #Added... verify if it works

'orientation', #Added, verify if it works. Angle btwn x-axis and major axis.

'MajorAxisLength',

'MinorAxisLength',

'Perimeter',

'MinIntensity',

'MeanIntensity',

'MaxIntensity']

output\_file = open('image\_measurements.csv', 'w')

output\_file.write('Grain #' + "," + "," + ",".join(propList) + '\n')

grain\_number = 1

for region\_props in regions:

output\_file.write(str(grain\_number) + ',')

#output cluster properties to the excel file

# output\_file.write(str(region\_props['Label']))

for i,prop in enumerate(propList):

if(prop == 'Area'):

to\_print = region\_props[prop]\*pixels\_to\_um\*\*2

elif(prop == 'orientation'):

to\_print = region\_props[prop]\*57.2958

elif(prop.find('Intensity') < 0):

to\_print = region\_props[prop]\*pixels\_to\_um

else:

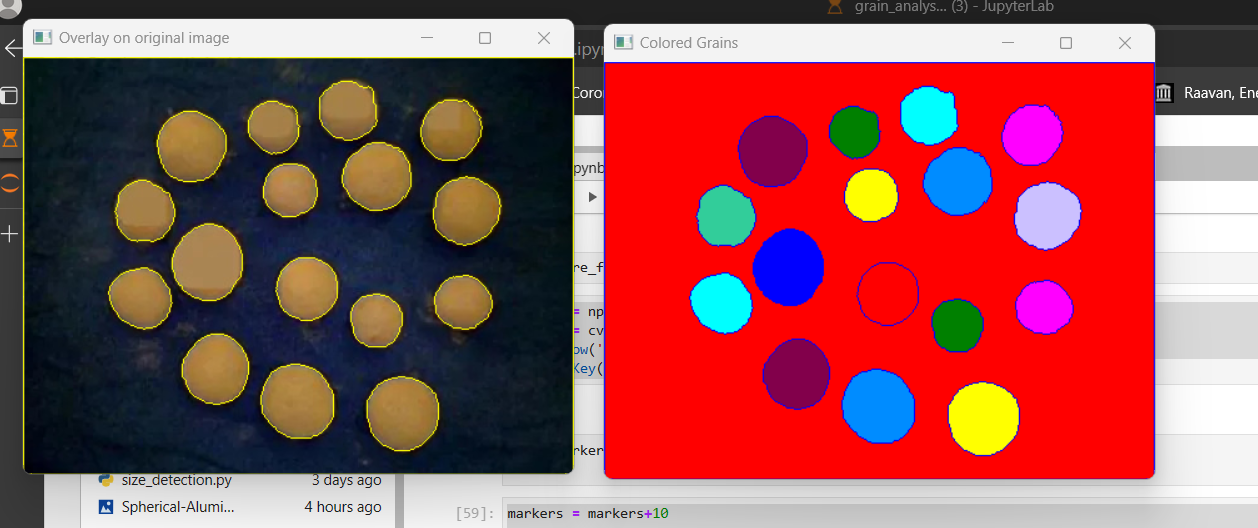
to\_print = region\_props[prop]

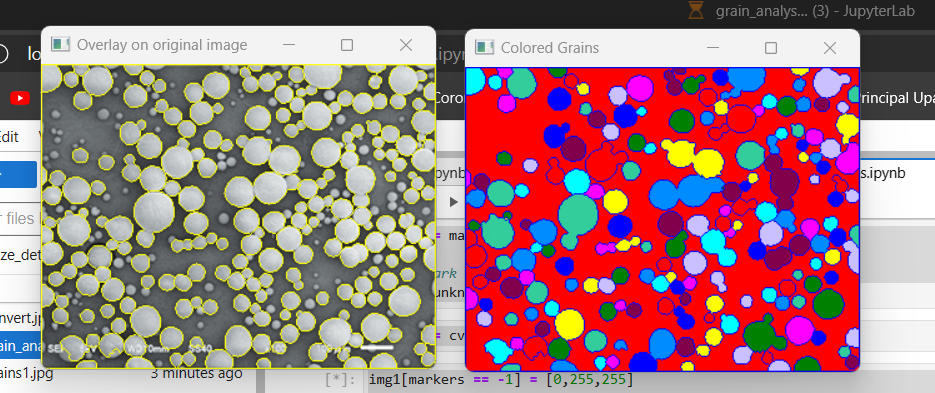
output\_file.write(',' + str(to\_print))

output\_file.write('\n')

grain\_number += 1

output\_file.close()

output:



**TASK2 :Satellite Image Analysis for Building and Tree Detection:**

Satellite Image Analysis for Building and Tree Detection with YOLOv8:

Satellite imagery has become an invaluable resource for various applications, including urban planning, environmental monitoring, and disaster response. One of the key challenges in satellite image analysis is the detection of specific objects such as buildings and trees, which are essential for understanding and managing land use and environmental changes. This documentation provides an overview of a project that focuses on using YOLOv8, a state-of-the-art object detection algorithm, for detecting buildings and trees in satellite images.

**Objective:**

The primary objective of this project is to develop a robust and accurate system for detecting buildings and trees in satellite images using YOLOv8. By leveraging deep learning techniques, we aim to automate the process of object detection, enabling efficient analysis of large-scale satellite datasets.

Methodology:

Data Collection:

1.Acquire satellite images from various sources, such as satellite imagery providers or public datasets.eg roboflow it provide diverse and different pre-annotated datasets.Ensure that the dataset contains diverse scenes encompassing urban and rural areas with different building and tree densities

2.user can annotate dataset using roboflow using advanced tools to reduce workload.

Preprocessing:

1.Resize and normalise the images to facilitate efficient training and inference with the YOLOv8 model.

Training YOLOv8:

1.Prepare annotated datasets where buildings and trees are labelled as distinct classes.

2.Utilize transfer learning to fine-tune the pre-trained YOLOv8 model on the annotated satellite image dataset.

3.Optimize training parameters, including learning rate, batch size, and number of epochs, to achieve optimal performance.

Validation and Testing:

1.Evaluate the trained YOLOv8 model on validation datasets to assess its accuracy and generalization capabilities.

2.Conduct comprehensive testing on independent test datasets to measure the model's performance in real-world scenarios.

**What is yolov8?**

The field of computer vision advances with the release of YOLOv8, a model that defines a new state of the art for object detection,instance segmentation and classification.Along with improvements to the model architecture itself, YOLOv8 introduces developers to a new friendly interface.YOLOv8 was developed by Ultralytics, YOLO(YOU ONLY LOOK ONCE) series of models has become famous in the computer vision world. YOLO's fame is attributable to its considerable accuracy while maintaining a small model size. YOLO models can be trained on a single GPU, which makes it accessible to a wide range of developers.

DATASETS:

The datasets were fetched from a known website called **ROBOFLOW.**

**Link:** [**https://roboflow.com/**](https://roboflow.com/)

* Sign up the website using a gmail account .after signing up choose appropriate options for detection and create project(working space)
* Upload your image dataset which are need to annotated and save them
* Annotate them using advanced tools and specify objects into classes.
* After annotation split annotated dataset into train,test and validation into (70,20,10)ratio.
* Download the annotated dataset in yolov8 format

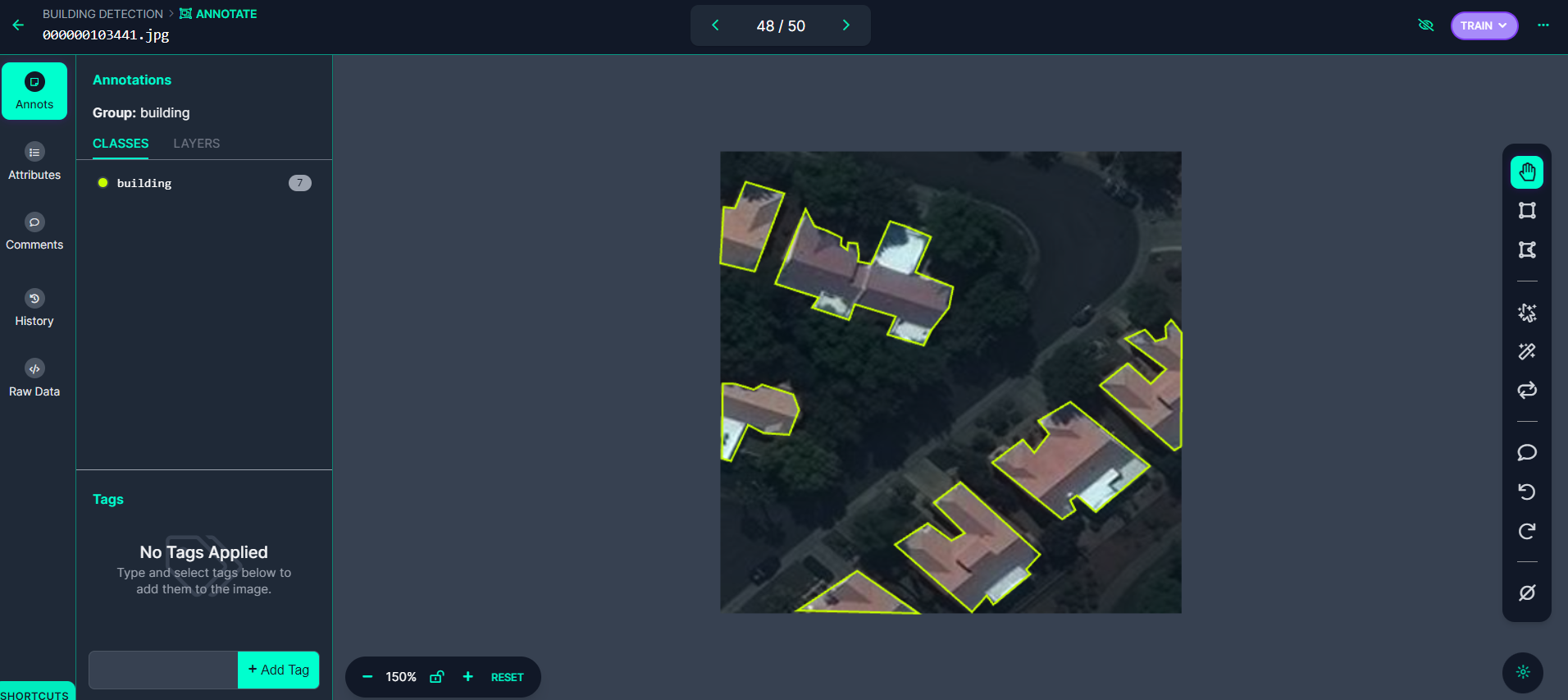
There are total 3 datasets used in these project:

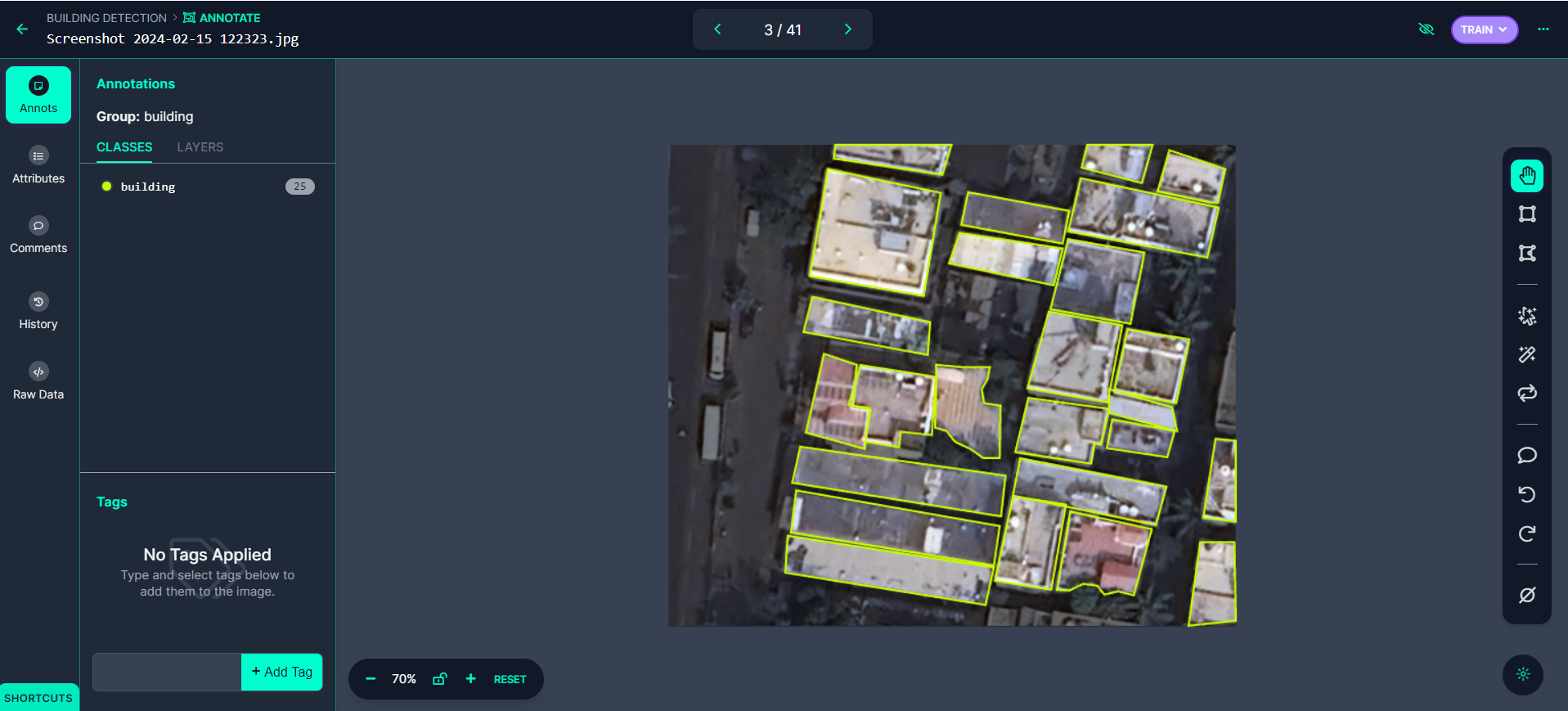
1.international dataset for buildings.

2.Indian dataset for buildings.

3.international dataset for trees.

International building annotation:



Indian building annotation:

Similarly, a dataset was created for trees.

**Models:**

YOLOv8 pretrained Detect models are shown here. Detect, Segment and Pose models are pretrained on the COCO dataset.

1.YOLOv8n

2.YOLOv8s

3.YOLOv8m

4.YOLOv8l

5.YOLOv8X

In this project YOLOv8s model is used and trained on the dataset using the model.Ultralytics package is used while training dataset

**Training:**

**#upload dataset on drive and mount google drive in google colab so that yser can easily fetch the data.**

from google.colab import drive

drive.mount('/content/drive')

**#make sure training is done in gpu for fast processing**

!nvidia-smi

import os

HOME = os.getcwd()

print(HOME)

**#install all required packages**

!pip install ultralytics

from IPython import display

display.clear\_output()

import ultralytics

ultralytics.checks()

from ultralytics import YOLO

from IPython.display import display, Image

**#giving path of dataset**

%cd /content/drive/MyDrive/international\_dataset\_building

**#training the dataset**

**#Make sure to mention path of train and validation dataset present in data.yaml file**

**#make sure the name of the dataset should have space or any invalid operators.**

!yolo task=detect mode=train model=yolov8s.pt data= data.yaml epochs=25 imgsz=224 plots=True

**#results are stored in run inside the dataset folder**

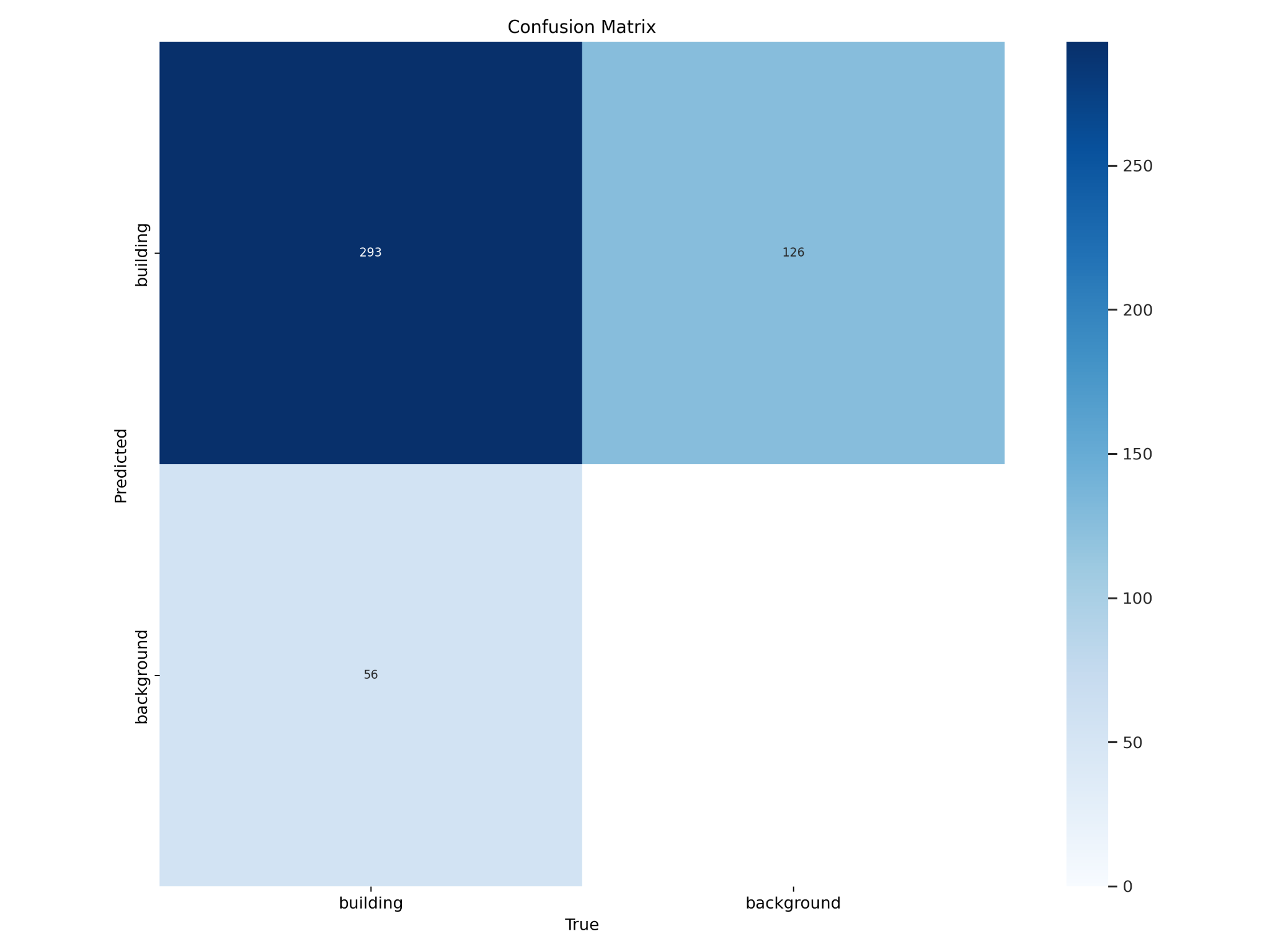
**Displaying result:**

!ls runs/detect/train2

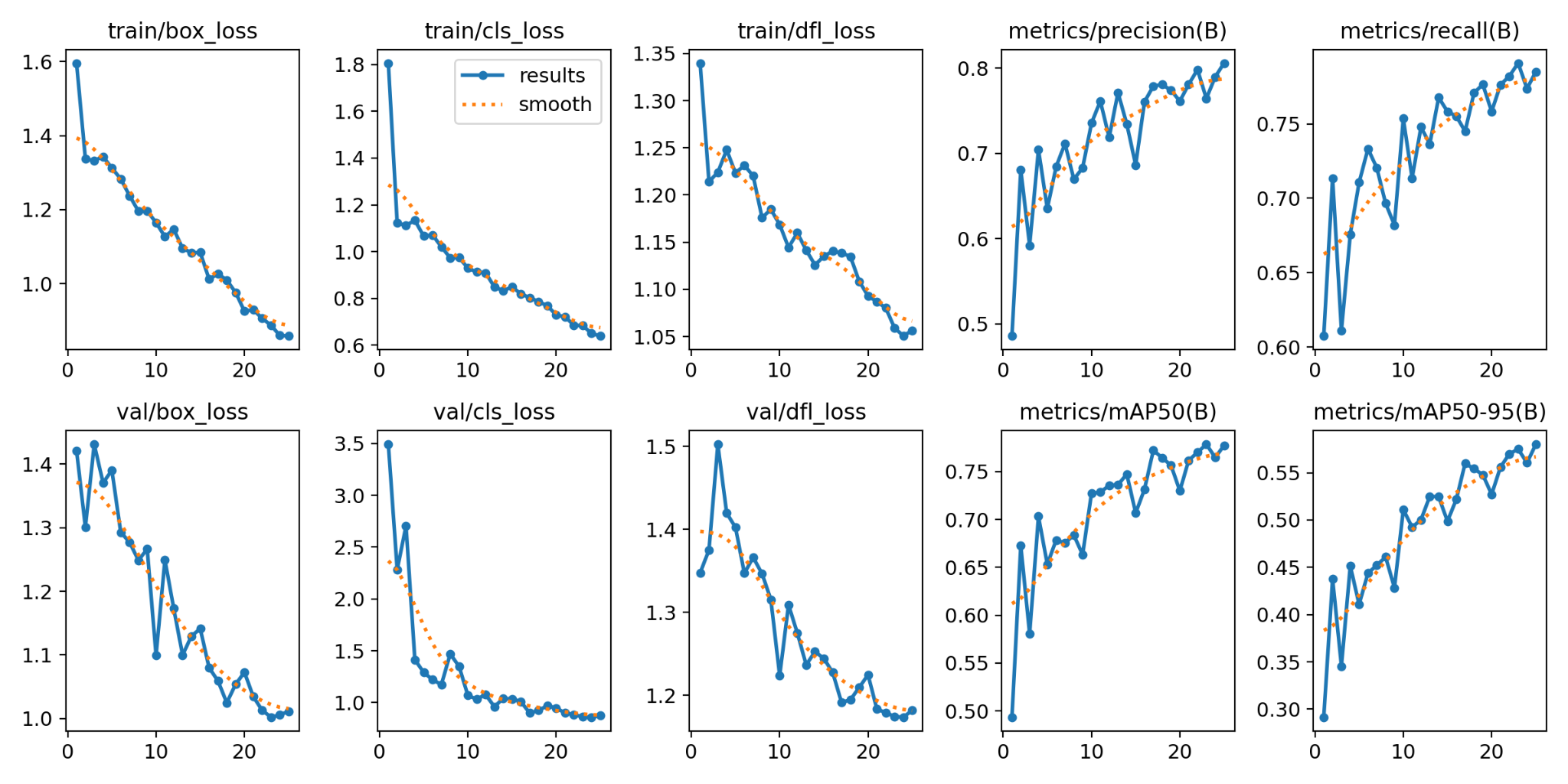
Image(filename='/content/drive/MyDrive/building\_detection\_yolov8/runs/detect/train2/confusion\_matrix.png', width=600)

Image(filename='/content/drive/MyDrive/building\_detection\_yolov8/runs/detect/train2/results.png', width=600).

**Confusion matrix:**



**Result:**

****

**Validation:**

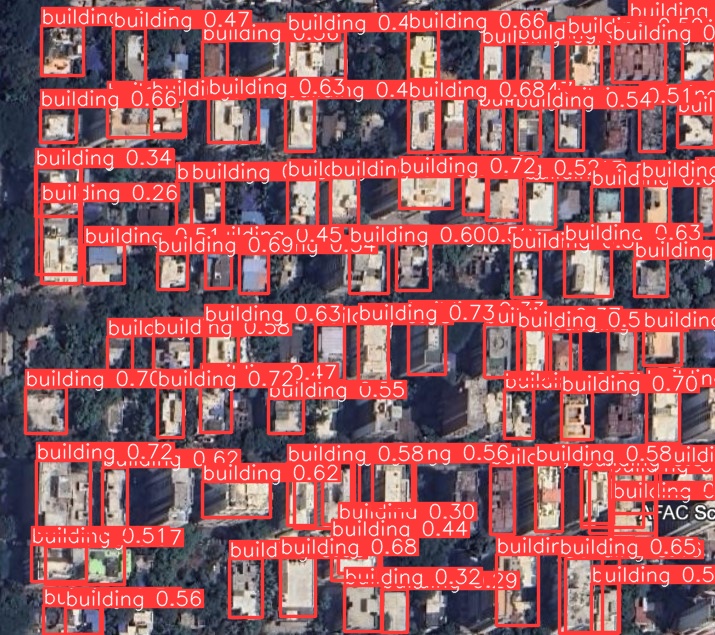
!yolo task=detect mode=val model=/content/drive/MyDrive/building\_detection\_yolov8/runs/detect/train2/weights/

**Testing:**

**#external image was given as input for testing**

!yolo task=detect mode=predict model=/content/drive/MyDrive/building\_detection\_yolov8/runs/detect/train2/weights/best.pt conf=0.25 source=/map2.jpg

**OUTPUT:**

****

Hence building were detected similarly trees were detected using same process.