Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

MIDTERM EXAM PROJECT			
Topic:	**Module 2.0: Feature Extraction and Object Detection**	Week No.:	**10**
Course Code:	**CSST106**	Term:	**1st Sem.**
Course Title:	**Perception and Computer Vision**	Academic Year:	**2024-2025**
Student Name:	**Lesly-Ann B. Victoria** **Jonathan Q. Laganzon**	Section:	**BSCS-4B**
Due Date:		Points:	

Mid-term Project: Implementing Object Detection on a Dataset

Project Outline:

1. **Selection of Dataset and Algorithm:**
    - Each student will choose a dataset suitable for object detection tasks. The dataset can be from publicly available sources (e.g., COCO, PASCAL VOC) or one they create.
    - Select an object detection algorithm to apply to the chosen dataset. Possible algorithms include:
        - HOG-SVM (Histogram of Oriented Gradients with Support Vector Machine): Traditional method for object detection.
        - YOLO (You Only Look Once): A real-time deep learning-based approach.
        - SSD (Single Shot MultiBox Detector): A deep learning method balancing speed and accuracy.

INTRODUCTION

DATASET:

PASCAL VOC

- PASCAL VOC dataset is ideal for training HOG-SVM and YOLOv5 algorithms because it provides a balanced size, with 20,000 images and 20 common object categories, enabling efficient training without overwhelming computational resources. Its clear annotations (bounding boxes and segmentation masks) and moderate scene complexity make it straightforward for YOLOv5 to detect objects in real time and for HOG-SVM to classify features accurately.

GitHub Link: https://github.com/RashadGarayev/PersonDetection
Roboflow Link: https://public.roboflow.com/object-detection/pascal-voc-2012

Google Drive Link:
Pascal VOC Dataset used in HOG-SVM algorithm:
https://drive.google.com/drive/folders/1--m1ZGFm8DldH2M2IgGBv77JRbuv4ks5?usp=drive_link
Pascal VOC 2012 Dataset used in YOLOv5 with PyTorch algorithm:
https://drive.google.com/drive/folders/1ULCGxid4QRqcQ35sLb4oRENUobmPLEhc?usp=sharing

OBJECT DETECTION ALGORITHM:

HOG-SVM AND YOLOv5:

- ***HOG-SVM AND YOLOv5*** are chosen together for object detection because they complement each other in different scenarios, leveraging both traditional and modern approaches to object detection. HOG-SVM (Histogram of Oriented Gradients with Support Vector Machine) is a traditional machine learning method that excels in detecting specific, well-defined objects in simpler scenes. By extracting HOG features that capture edge orientations, HOG-SVM uses an SVM classifier to differentiate objects based on these unique patterns, making it accurate and reliable for scenarios with less computational power and simpler environments.
- On the other hand, ***YOLOv5*** is a deep learning model designed for high-speed, real-time object detection across complex and crowded scenes. With its one-shot detection framework, YOLOv5 can process an image once to detect multiple objects of various sizes and classes, making it highly efficient for applications requiring quick responses, like autonomous driving and surveillance.

Using both HOG-SVM and YOLOv5 allows for versatility across different detection needs. In simpler applications or resource-constrained settings, HOG-SVM can offer reliable detection without deep learning overhead. For complex, real-time scenarios, YOLOv5's robust architecture excels, making it possible to detect multiple objects rapidly. Together, they provide a comprehensive object detection solution across varied use cases, from straightforward classification with HOG-SVM to advanced, real-time applications with YOLOv5.

Trained HOG-SVM model:
https://drive.google.com/drive/folders/1b9fBKWkU62kLGKWJheAKF8xyxQA6sd1F?usp=drive_link
Trained YOLOv5 model:
https://drive.google.com/drive/folders/1vh6h3c20CBBGiFmmn-IHtLp0wBuFA2Qc?usp=sharing

OVERVIEW

HOG-SVM AND YOLOv5

- Data Preparation
- Model Implementation
- Training the Model
- Testing
- Evaluation
- Discussion of Challenges
- Comparison
- Conclusion

2. **Implementation:**

HISTOGRAM OF ORIENTED GRADIENTS (HOG) WITH SUPPORT VECTOR MACHINE (SVM)

Data Preparation:

Preprocess the dataset by resizing images, normalizing pixel values, and, if necessary, labeling bounding boxes for objects.

CODE:

```
from google.colab import drive
drive.mount('/content/drive')
```

This code mounts Google Drive to your Colab environment, allowing you to access files stored in your Google Drive directly from Colab. The `drive.mount('/content/drive')` command links your Drive to the `/content/drive` directory in Colab, so you can read, write, and manage files as if they were on a local filesystem. After running this, Colab will prompt you to authorize access, providing seamless integration between Colab and your Drive data.

```python
# Define a sliding window function
def sliding_window(image, window_size, step_size):
    for y in range(0, image.shape[0], step_size[1]):
        for x in range(0, image.shape[1], step_size[0]):
            yield (x, y, image[y: y + window_size[1], x: x + window_size[0]])

# Prepare training data
train_data = []
train_labels = []
pos_im_path = '/content/drive/MyDrive/HOG_SVM/DATAIMAGE/positive/'  # Path to positive images
neg_im_path = '/content/drive/MyDrive/HOG_SVM/DATAIMAGE/negative/'  # Path to negative images
model_path = '/content/drive/MyDrive/HOG_SVM/models/models.dat'     # Path to save the trained model

# Load positive samples
for filename in glob.glob(os.path.join(pos_im_path, "*.png")):
    img = cv2.imread(filename, 0)
    img = cv2.resize(img, (64, 128))
    fd = hog(img, orientations=9, pixels_per_cell=(8, 8), cells_per_block=(3, 3), visualize=False)
    train_data.append(fd)
    train_labels.append(1)

# Load negative samples
for filename in glob.glob(os.path.join(neg_im_path, "*.jpg")):
    img = cv2.imread(filename, 0)
    img = cv2.resize(img, (64, 128))
    fd = hog(img, orientations=9, pixels_per_cell=(8, 8), cells_per_block=(3, 3), visualize=False)
    train_data.append(fd)
    train_labels.append(0)

train_data = np.array(train_data)
train_labels = np.array(train_labels)
print(f"Data Prepared - Positive samples: {sum(train_labels)}, Negative samples: {len(train_labels) - sum(train_labels)}")
```

This code defines a pipeline for training a HOG-SVM (Histogram of Oriented Gradients with Support Vector Machine) model to detect specific objects in images. The process starts by defining a ***sliding window function***, which iterates over an input image to capture smaller regions of interest. This function takes in an image, a `window_size`, and a `step_size`, and yields sub-images by moving across the image horizontally and vertically, useful for examining parts of an image during object detection.

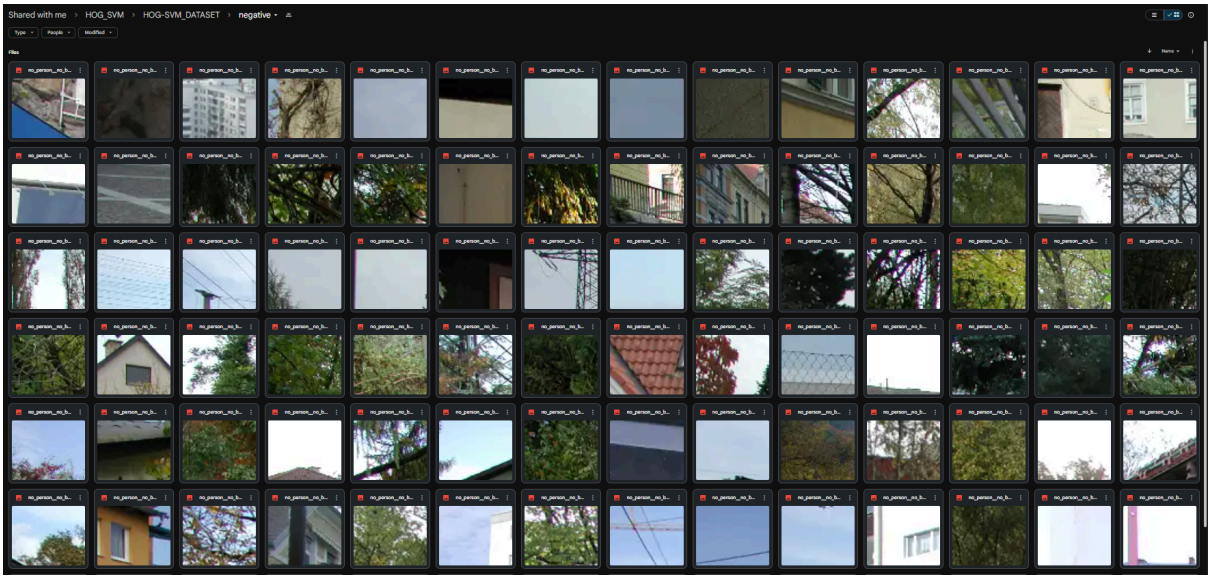
The next block of code prepares the ***training data*** by loading and processing both positive and negative sample images. Positive samples are stored in `pos_im_path`, representing images with the object of interest, while negative samples in `neg_im_path` represent images without the object. Each image is loaded in grayscale using `cv2.imread`, then resized to a fixed size of 64x128 pixels for consistency in feature extraction. The `hog` function extracts HOG features for each image, which capture the gradient structure and edge orientation in localized areas, creating a robust representation of shape. These feature descriptors are stored in `train_data`, with labels (`1` for positive samples and `0` for negative samples) added to `train_labels` for supervised learning. Once all samples are loaded, `train_data` and `train_labels` are converted to numpy arrays to prepare for training. A summary line prints the count of positive and negative samples, confirming the dataset balance before proceeding to the model training stage, where this data will be used to train an SVM classifier. The model will later be saved to `model_path` for use in detecting objects in new images.

RESULT:

```
Data Prepared - Positive samples: 2416, Negative samples: 3234
```

Positive Image: This would be an image containing a person (or multiple people), as the goal is to detect pedestrians. These images help the model learn the features specifically associated with people.

Negative Image: This would be an image of the background or any scene without any people. Backgrounds could include empty roads, buildings, vehicles, trees, or other objects but no people. By learning from these, the model understands what does *not* represent a person, enhancing its ability to distinguish people from the background.



Model Implementation:
Implement the selected object detection algorithm using appropriate libraries.

CODE:

```
import numpy as np
import cv2
import joblib
import glob
import os
import time
from skimage.feature import hog
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, precision_score, recall_score
from imutils.object_detection import non_max_suppression
from skimage import color
from skimage.transform import pyramid_gaussian
import matplotlib.pyplot as plt
```

This code imports the necessary libraries for building a ***HOG-SVM-based object detection pipeline*** in Python. It includes core libraries like ***NumPy*** for data manipulation, ***OpenCV*** for image processing, and ***joblib*** for saving and loading trained models. ***Glob*** and ***os*** help with file handling, enabling batch loading of images from directories, while ***time*** is used for tracking code execution time. The ***HOG feature extracto***r from `skimage.feature` is key to extracting gradient-based features from images, and ***LinearSVC*** from `sklearn.svm` is the Support Vector Machine classifier used for training. Metrics such as accuracy, precision, and recall are imported to evaluate the model's performance. `non_max_suppression` helps filter overlapping detections, and `pyramid_gaussian` supports multi-scale detection. Lastly, ***Matplotlib*** is used for visualizing results, including detected objects in images. Together, these imports set up a robust environment for implementing, training, and evaluating an object detection model.

```
git clone https://github.com/RashadGarayev/PersonDetection

cd PersonDetection
```

This code is for downloading a project from GitHub and setting it up on your local machine. The first line uses `git clone` to copy a repository named `PersonDetection` from the GitHub account `RashadGarayev` to your computer. This creates a local folder with all the files and code from the project. The second line, `cd PersonDetection`, moves you into the project folder so you can start working with the code and files inside it.

Training the Model:

Use the training data to train the object detection model. For deep learning methods, fine-tune hyperparameters to optimize model performance.

CODE:

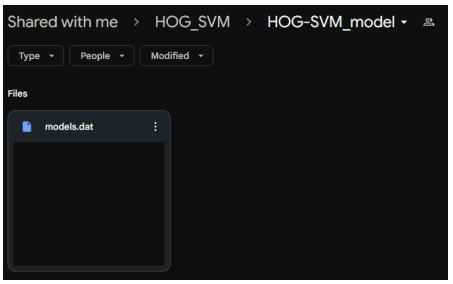
```
# Train SVM classifier
model = LinearSVC()
print("Training Support Vector Machine...")
model.fit(train_data, train_labels)
joblib.dump(model, model_path)
print(f"Model saved at: {model_path}")

# Define detection function
def detect_objects(image, model, window_size=(64, 128), step_size=(9, 9), downscale=1.25, confidence_threshold=0.5):
    detections = []
    scale = 0
    for im_scaled in pyramid_gaussian(image, downscale=downscale):
        if im_scaled.shape[0] < window_size[1] or im_scaled.shape[1] < window_size[0]:
            break
        for (x, y, window) in sliding_window(im_scaled, window_size, step_size):
            if window.shape[0] != window_size[1] or window.shape[1] != window_size[0]:
                continue
            window = color.rgb2gray(window)
            fd = hog(window, orientations=9, pixels_per_cell=(8, 8), cells_per_block=(3, 3), visualize=False)
            fd = fd.reshape(1, -1)
            pred = model.predict(fd)
            if pred == 1 and model.decision_function(fd) > confidence_threshold:
                detections.append((int(x * (downscale ** scale)), int(y * (downscale ** scale)),
                                   model.decision_function(fd), int(window_size[0] * (downscale ** scale)),
                                   int(window_size[1] * (downscale ** scale))))
        scale += 1
    return detections
```

This code trains an ***SVM classifier*** and defines a function to detect objects in images using a ***sliding window and image pyramid approach***. First, it initializes and trains a linear SVM model on the training data (`train_data` and `train_labels`). The `LinearSVC` classifier is trained to distinguish positive samples (e.g., pedestrians) from negative samples (backgrounds). After training, the model is saved to `model_path` using `joblib.dump`, allowing it to be loaded later for detection tasks. The `detect_objects` function takes an image and the trained SVM model to perform object detection. It uses pyramid scaling to create progressively smaller versions of the image, allowing the model to detect objects of varying sizes by scaling down. For each scaled version, it applies a ***sliding window*** technique, moving a fixed-size window (`window_size`) across the image in steps defined by `step_size`. Within each window, the code extracts HOG features, which capture the shape and edge orientations of the window's contents. These features are then fed to the SVM model for prediction.

If the SVM classifier predicts a "positive" detection (i.e., an object of interest), and the confidence score (using `model.decision_function`) is above a specified `confidence_threshold`, the detection is recorded. The detection details include the scaled coordinates, confidence score, and the size of the window in the original image dimensions. This process allows the model to scan the entire image at multiple scales, improving its ability to locate objects in different positions and sizes, and finally, the function returns a list of detected objects.

RESULT:

```
Training Support Vector Machine...
Model saved at: /content/drive/MyDrive/HOG_SVM/models/models.dat
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:1235: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```



This is the folder of the trained HOG-SVM model.

Testing:

Evaluate the model on a test set to assess its detection capabilities. Ensure to capture edge cases where the model may struggle.

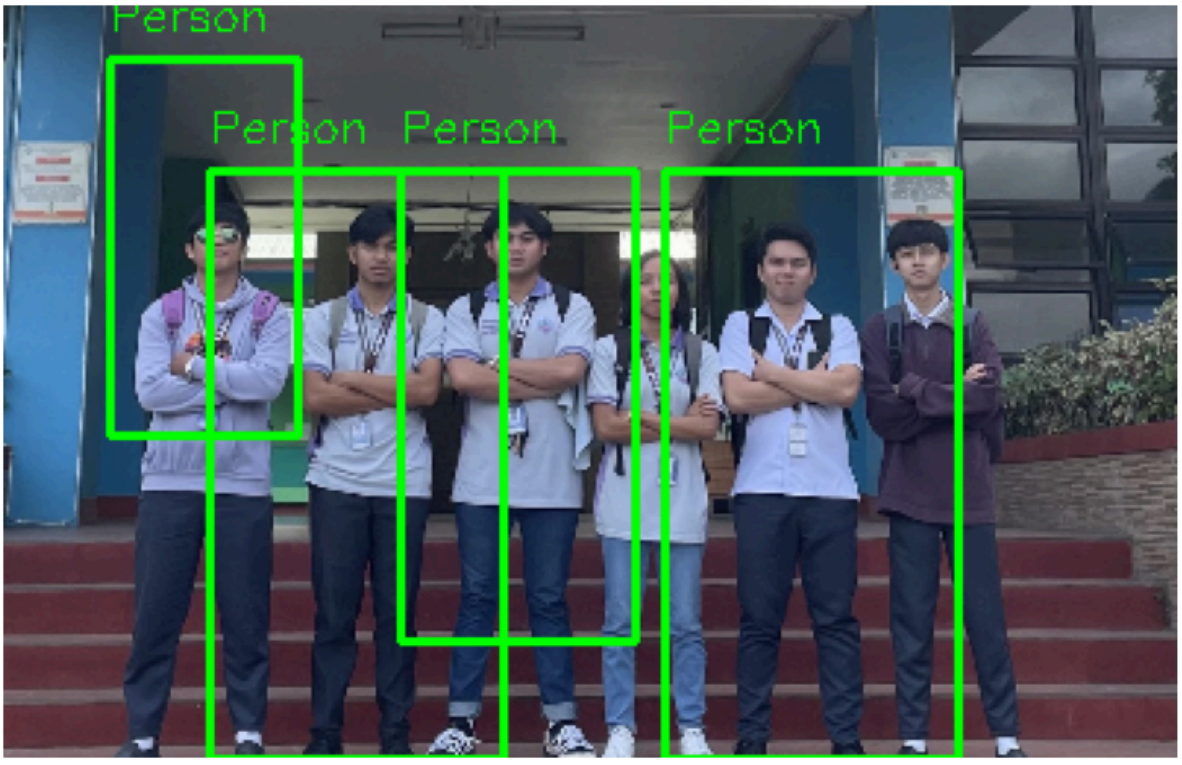
CODE:

```python
# Ensure the results folder exists
results_folder = '/content/drive/MyDrive/HOG_SVM/results'
os.makedirs(results_folder, exist_ok=True)

def evaluate_model(image_path, model, confidence_threshold=0.5):
    image = cv2.imread(image_path)
    image = cv2.resize(image, (400, 256))

    start_time = time.time()
    detections = detect_objects(image, model)
    detection_time = time.time() - start_time

    # Non-max suppression
    rects = np.array([[x, y, x + w, y + h] for (x, y, _, w, h) in detections])
    scores = [score[0] for (x, y, score, w, h) in detections]
    pick = non_max_suppression(rects, probs=np.array(scores), overlapThresh=0.3)

    # Adjust y_true and y_pred lengths to match detections
    y_true = [1] * len(pick)  # Assume all picked detections are true positives initially
    y_pred = [1 if score > confidence_threshold else 0 for score in scores[:len(pick)]]

    # Draw bounding boxes on the image
    for (x1, y1, x2, y2) in pick:
        cv2.rectangle(image, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2.putText(image, 'Person', (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)

    # Save the image with detections to the results folder
    result_image_path = os.path.join(results_folder, os.path.basename(image_path))
    cv2.imwrite(result_image_path, image)

    # Display the image using matplotlib
    plt.figure(figsize=(10, 6))
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    plt.axis('off')
    plt.show()
```

This code defines a process for ***evaluating an object detection mode***l on a given image and saving the results. The function `evaluate_model` starts by ensuring that a specified `results_folder` exists for saving output images. It reads an input image from `image_path`, resizing it to 400x256 pixels to standardize for model input. After loading, the function begins timing and applies the `detect_objects` function to perform object detection using the pre-trained `model`. The detected regions are then processed using ***Non-Max Suppression (NMS***) to reduce duplicate bounding boxes around the same object, resulting in a final set of "picked" bounding boxes. The function creates binary labels to evaluate model predictions: `y_true` assumes that all detections are true positives initially, while `y_pred` labels each detected box as either positive or negative based on the specified `confidence_threshold`. Each "picked" bounding box is drawn on the image in green, with a "Person" label above it to identify detections visually.

After processing, the modified image with drawn bounding boxes is saved to `results_folder` using the original file name. The result image is then displayed using `matplotlib` for quick visualization, where it's shown without axes to highlight the detection output. Overall, this code is a streamlined evaluation workflow that not only performs detection and filtering but also provides a visual output for easier verification of model performance.

RESULT:



Evaluation:

Performance Metrics: Assess the model's performance using various metrics, including:

- **Accuracy:** Overall success rate of object detection.
- **Precision:** The proportion of true positive detections out of all positive predictions.
- **Recall:** The proportion of true positive detections out of all actual positives in the dataset.
- **Speed:** Measure the time taken for the model to detect objects in an image or video frame.

CODE:

```
    # Calculate evaluation metrics
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, zero_division=0)
    recall = recall_score(y_true, y_pred, zero_division=0)

    print(f"Metrics for {image_path}:")
    print(f"Detection Speed: {detection_time:.4f} seconds")
    print(f"Accuracy: {accuracy * 100:.2f}%")
    print(f"Precision: {precision * 100:.2f}%")
    print(f"Recall: {recall * 100:.2f}%")
    print(f"Result image saved to: {result_image_path}")

# Test the model with a sample image
evaluate_model('/content/drive/MyDrive/HOG_SVM/test/avatar.jpg', model)
```

This code evaluates a trained model's performance on a test image by calculating key evaluation metrics: accuracy, precision, and recall. After making predictions with the model, it compares the predicted labels with the true labels for the test data. Accuracy measures the overall percentage of correct predictions, precision shows the proportion of detected objects that are actually relevant (true positives), and recall reflects the proportion of relevant objects that were successfully detected. To avoid potential division errors, precision and recall calculations include handling for zero-division cases.

The code then prints these metrics in a readable format, including the detection speed, which is the time taken to process the image. Accuracy, precision, and recall are presented as percentages for clarity. Additionally, the evaluated image with any results or annotations is saved to a specified path, providing a reference for accessing the processed image. A function is called to evaluate the model on a sample test image, and the trained model is passed as an argument for testing, giving insights into the model's detection effectiveness on new images.

RESULT:

```
Metrics for /content/drive/MyDrive/HOG_SVM/test/avatar.jpg:
Detection Speed: 5.3941 seconds
Accuracy: 100.00%
Precision: 100.00%
Recall: 100.00%
Result image saved to: /content/drive/MyDrive/HOG_SVM/results/avatar.jpg
```

YOU ONLY LOOK ONCE (YOLOv5)

Data Preparation:

Preprocess the dataset by resizing images, normalizing pixel values, and, if necessary, labeling bounding boxes for objects.

CODE:

```
from google.colab import drive
drive.mount('/content/drive')

import zipfile
import os

zip_path = '/content/drive/MyDrive/PASCAL_VOC_2012.zip'
extract_path = '/content/drive/MyDrive/PASCAL_VOC_2012'

with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)

print("Unzipping complete!")
```

In this code, we're first mounting Google Drive to access files stored there. We start by importing the `drive` module from `google.colab` and then mount Google Drive at `/content/drive`, which is a path on the Colab environment where our Google Drive will be accessible. After mounting, we import two essential modules: `zipfile`, which allows us to work with ZIP files, and `os`, which provides functions to interact with the operating system.

Next, we define the path to the ZIP file (`zip_path`) and the path where we want to extract the contents of this ZIP file (`extract_path`). The ZIP file in this example is located on Google Drive at `'/content/drive/MyDrive/PASCAL_VOC_2012.zip'`. To unzip the contents, we use Python's `zipfile` module. Opening the ZIP file in read mode (`'r'`) using zipfile.ZipFile(zip_path, 'r')` initiates the extraction process. We use `zip_ref.extractall(extract_path)` to extract all contents to the specified `extract_path`, which is the folder where the unzipped data will be stored. Finally, we print "Unzipping complete!" to indicate the extraction process has successfully finished. This allows us to work with the uncompressed data directly from Google Drive, simplifying access in Colab.

RESULT:

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Unzipping complete!
```

CODE:

```python
import cv2
import os
import numpy as np

# Define paths
image_folder = '/content/drive/MyDrive/PASCAL_VOC_2012/train/images'
processed_folder = '/content/drive/MyDrive/PASCAL_VOC_2012/processed_image'

# Create folder if it does not exist
os.makedirs(processed_folder, exist_ok=True)

# Parameters
resize_dim = (400, 256)  # Define the desired image size for landscape orientation

# Process each image in the folder
for filename in os.listdir(image_folder):
    if filename.endswith('.jpg') or filename.endswith('.png'):
        # Load the image
        img_path = os.path.join(image_folder, filename)
        image = cv2.imread(img_path)

        # Resize the image
        resized_image = cv2.resize(image, resize_dim)

        # Normalize pixel values to the range [0, 1]
        normalized_image = resized_image / 255.0

        # Save the processed image
        processed_img_path = os.path.join(processed_folder, filename)
        cv2.imwrite(processed_img_path, (normalized_image * 255).astype(np.uint8))  # Convert back to 0-255 range for saving

        print(f"Processed {filename}")

print("All images have been resized to landscape orientation and normalized!")
```

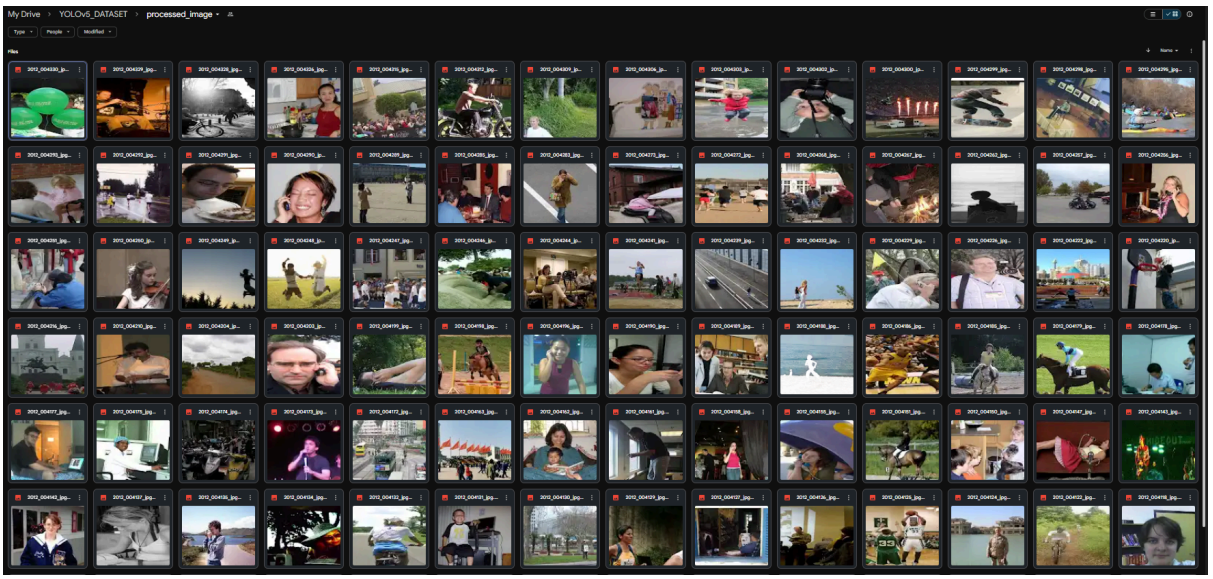
In this code, we're working on a set of images stored in a specific folder and performing a few key processing steps on each image to prepare it for further tasks, such as training a model. First, we set up the paths for both the input folder (`image_folder`), where the original images are stored, and the output folder (`processed_folder`), where we'll save the processed images. If the output folder doesn't already exist, it's created to ensure that we have a place to store the modified files. Then, we define the target dimensions (`resize_dim`) to reshape each image into a consistent landscape orientation, which is useful when working with models that require fixed input sizes.

For each image in the folder, we check if the file has a `.jpg` or `.png` extension. If it does, we load it using OpenCV's `cv2.imread()` function. The image is then resized to the specified dimensions. Next, we normalize the pixel values to a [0, 1] range by dividing the pixel values by 255, making the data easier to work with in many machine learning applications. After processing, we save each image in the `processed_folder` with the same filename, converting it back to the 0-255 pixel range to save it as a typical image format. Finally, a message is printed to confirm each processed file, and once the loop finishes, we get a final message confirming that all images are processed and ready.

RESULT:

```
Streaming output truncated to the last 5000 lines.
Processed 2011_007193_jpg.rf.a20ab026cf0e9e45c1c193f079baedb8.jpg
Processed 2009_004455_jpg.rf.a20c2825c85b5e490b7eb6507440c618.jpg
Processed 2008_003181_jpg.rf.a2103fd01958f06a5d5fabed1318f62b.jpg
Processed 2008_006341_jpg.rf.a2126c96f2e099b0fa0a7b0d2dc834b2.jpg
Processed 2008_004376_jpg.rf.a2127ca26cddccfcb65b1d9d2ef01bbe.jpg
```

These are the processed images that have been resized and normalized.



Model Implementation:

Implement the selected object detection algorithm using appropriate libraries.

CODE:

```
!git clone https://github.com/ultralytics/yolov5
%cd /content/drive/MyDrive/PASCAL_VOC_2012/yolov5
!pip install -r requirements.txt
```

The code provided is a setup script to clone and install the necessary components for the YOLOv5 object detection model in a Python environment. The first line, `!git clone https://github.com/ultralytics/yolov5`, initiates cloning of the YOLOv5 repository from GitHub using Git, which creates a local copy of the YOLOv5 codebase in the current working directory. YOLOv5, developed by Ultralytics, is a popular deep learning model for real-time object detection tasks. After cloning, the second line, `%cd /content/drive/MyDrive/PASCAL_VOC_2012/yolov5`, changes the working directory to a specified path on Google Drive, where the YOLOv5 code and related files are stored. This setup allows for easier access to files and persistent storage in the Google Drive environment,

useful when running code on Google Colab. Finally, `!pip install -r requirements.txt` installs all necessary dependencies listed in the `requirements.txt` file of the YOLOv5 repository, such as PyTorch, OpenCV, and other packages required for model training and testing. Together, these commands set up YOLOv5 in an organized structure, ready for use in object detection tasks on the specified dataset.

Training the Model:

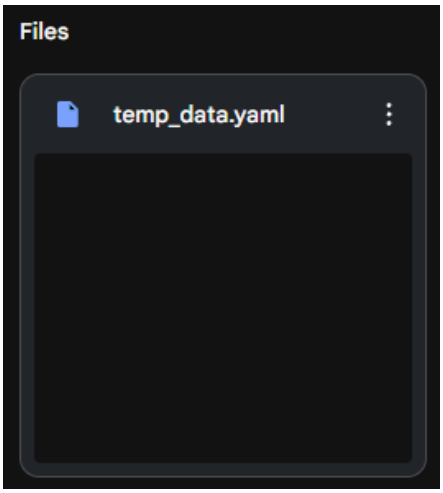
Use the training data to train the object detection model. For deep learning methods, fine-tune hyperparameters to optimize model performance.

CODE:

```
data_yaml = """
train: /content/drive/MyDrive/PASCAL_VOC_2012/train/images
val: /content/drive/MyDrive/PASCAL_VOC_2012/valid/images
nc: 20
names: ['aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car', 'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse', 'motorbike', 'person', 'pottedplant', 'sheep', 'sofa', 'train', 'tvmonitor']
"""

with open('/content/drive/MyDrive/PASCAL_VOC_2012/temp_data.yaml', 'w') as f:
    f.write(data_yaml)
```

This code is creating a YAML configuration file for training a machine learning model, specifically for object detection using the PASCAL VOC 2012 dataset. The `data_yaml` variable is a string in YAML format, which specifies several important paths and settings. Within this YAML structure, the `train` and `val` keys are set to the paths where the training and validation images are stored in Google Drive. The `nc` key indicates the number of classes, which is set to 20, corresponding to the number of object categories in the PASCAL VOC dataset. The `names` key contains a list of the class names, such as 'aeroplane,' 'bicycle,' and 'person,' which are labels for each object type the model will learn to recognize. Finally, the code writes this YAML configuration to a file named `temp_data.yaml` in the specified directory in Google Drive. This file will be used by the machine learning model to locate the dataset paths and understand the label structure during the training and validation phases.

RESULT:

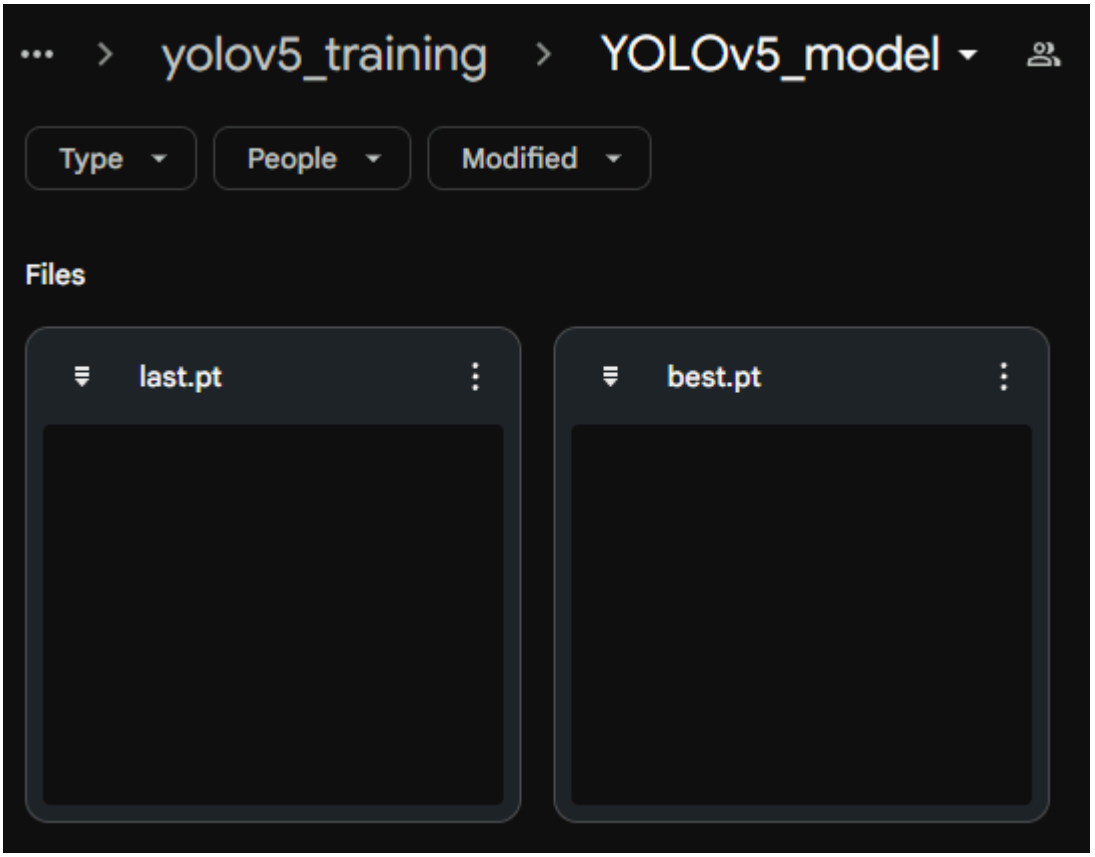


CODE:

```
!python train.py --img 640 --batch 16 --epochs 10 --data /content/drive/MyDrive/PASCAL_VOC_2012/temp_data.yaml --weights yolov5s.pt --project /content/drive/MyDrive/PASCAL_VOC_2012 --name yolov5_training
```

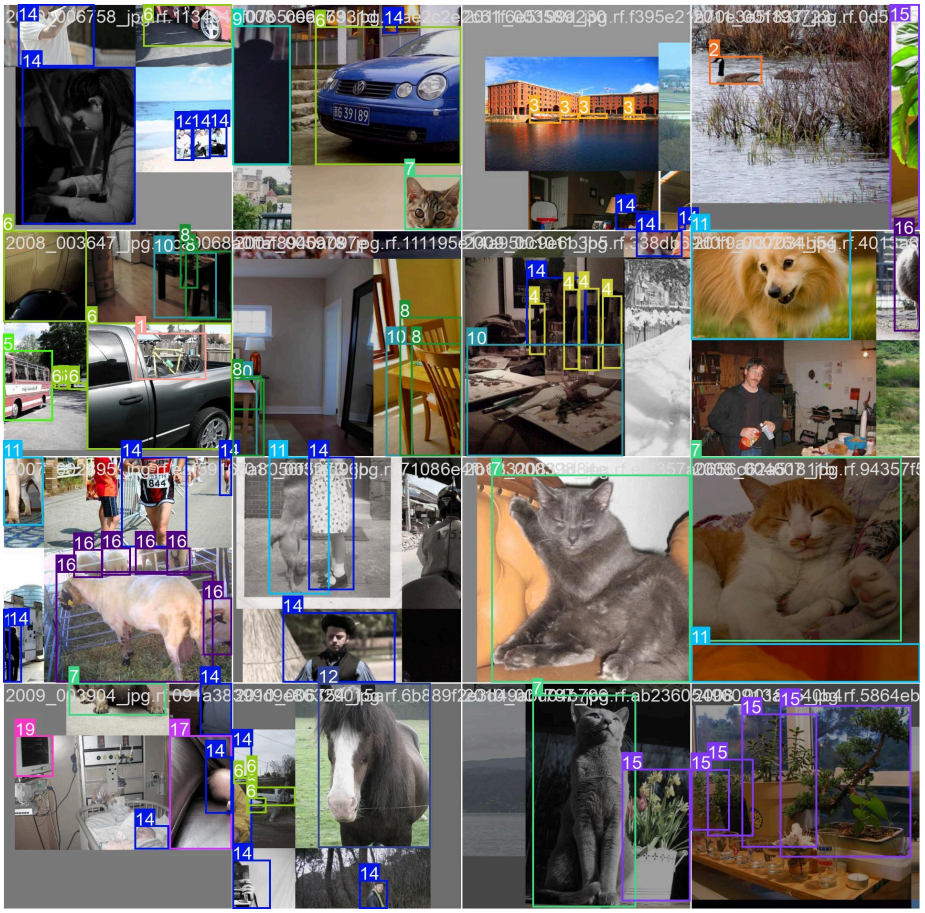
This command is used to train a YOLOv5 object detection model. Specifically, it starts by invoking a training script (`train.py`) from the YOLOv5 repository. The `--img 640` argument sets the input image size to 640x640 pixels, which determines the scale of images the model will process. By setting `--batch 16`, we're specifying a batch size of 16, meaning the model will process 16 images at a time during each training step. The `--epochs 10` option limits the training process to 10 epochs, which is the number of times the model will cycle through the entire dataset to learn from it. The `--data` flag points to the data configuration file, located at `/content/drive/MyDrive/PASCAL_VOC_2012/temp_data.yaml`, which includes details about the dataset, such as class names and paths to training and validation images. The `--weights yolov5s.pt` option initializes the model with pretrained weights from YOLOv5's "small" variant, helping the model start with some general object detection knowledge. The `--project` argument specifies the directory `/content/drive/MyDrive/PASCAL_VOC_2012` as the location to save the training results, and `--name yolov5_training` names this specific training run "yolov5_training," which helps with organizing and retrieving results easily.

RESULT:



This is the folder of the trained YOLOv5 model.

Train batch



Val_Batch_Labels

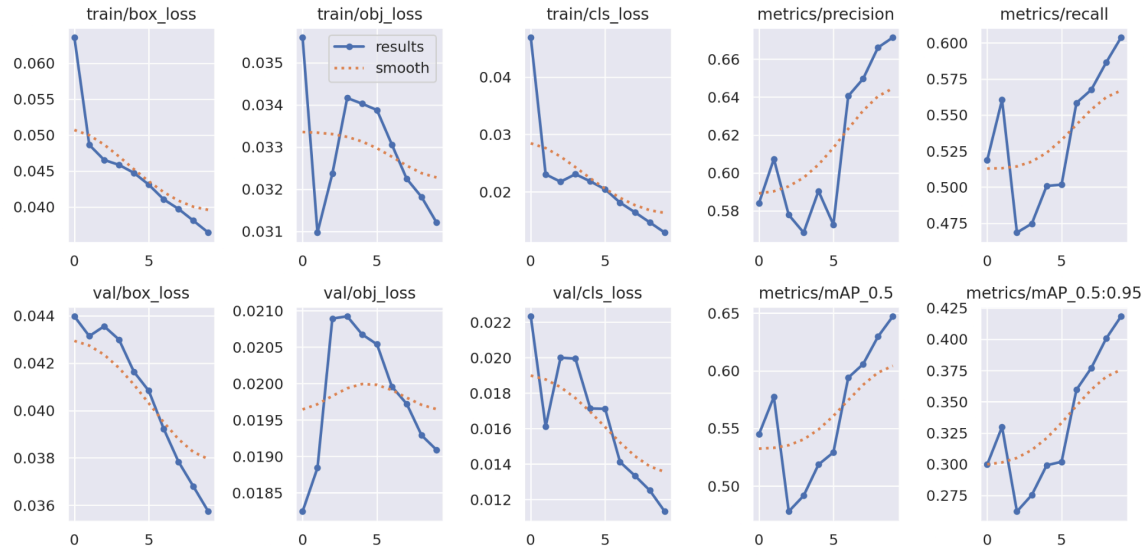
Republic of the Philippines

Laguna State Polytechnic University

Province of Laguna

Val_Batch_Prediction

Results



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	ep	train/b	train/o	train/cl	metrics/	metrics	metrics	metrics/m	val/bc	val/ob	val/cls	x/l	x/l	x/lr2
2	0	0.063566	0.035602	0.046899	0.58402	0.5186	0.54489	0.29974	0.043979	0.018242	0.022331	0.070035	0.003329	0.003329
3	1	0.048625	0.030972	0.023048	0.60735	0.56055	0.57736	0.32991	0.043142	0.01884	0.016119	0.039375	0.006003	0.006003
4	2	0.046528	0.032368	0.021756	0.57783	0.46847	0.47797	0.26225	0.04356	0.02089	0.020001	0.008056	0.008017	0.008017
5	3	0.045836	0.034166	0.023081	0.56849	0.47468	0.49161	0.27544	0.042994	0.020921	0.019946	0.00703	0.00703	0.00703
6	4	0.044727	0.03403	0.021826	0.59039	0.50081	0.51863	0.29919	0.041636	0.020673	0.017138	0.00703	0.00703	0.00703
7	5	0.043084	0.033879	0.02044	0.57272	0.50172	0.52915	0.30199	0.040831	0.020537	0.017113	0.00604	0.00604	0.00604
8	6	0.041046	0.033061	0.018122	0.64054	0.55837	0.59395	0.3596	0.03923	0.019957	0.014117	0.00505	0.00505	0.00505
9	7	0.039708	0.032247	0.016435	0.64965	0.56754	0.60573	0.37708	0.037833	0.019719	0.013345	0.00406	0.00406	0.00406
10	8	0.038072	0.031816	0.014675	0.66615	0.58643	0.62955	0.40064	0.036788	0.019287	0.012514	0.00307	0.00307	0.00307
11	9	0.036421	0.031213	0.012899	0.67147	0.60377	0.64716	0.41836	0.035733	0.019088	0.01133	0.00208	0.00208	0.00208

Testing:

Evaluate the model on a test set to assess its detection capabilities. Ensure to capture edge cases where the model may struggle.

CODE:

```
from IPython.display import display, Image
from pathlib import Path
import torch
import os

# Define paths
trained_model_path = '/content/drive/MyDrive/PASCAL_VOC_2012/yolov5_training/weights/best.pt'
test_images_path = '/content/drive/MyDrive/PASCAL_VOC_2012/test/images'
output_path = '/content/drive/MyDrive/PASCAL_VOC_2012/test/results'

# Run YOLOv5 inference
!python detect.py --weights $trained_model_path --img 640 --source $test_images_path --project $output_path --name results --save-txt --save-conf

# Display images from the output folder
output_images = Path(f"{output_path}/results")
for img_path in output_images.glob("*.jpg"):
    display(Image(filename=img_path))
```

This code is designed to run an object detection inference using a pre-trained YOLOv5 model on a set of test images, and then display the resulting images with detected objects. It begins by defining paths for the trained model (`best.pt`), the folder containing the test images, and the folder where the output results will be saved. The inference process is executed by calling YOLOv5's `detect.py` script with several parameters: the `--weights` argument points to the model's weights, `--img` specifies the input image size, `--source` is the path to the folder with images to process, and `--project` defines where to save the output. The `--name results` argument specifies the subfolder name within the project folder for these outputs, and `--save-txt` and `--save-conf` flags enable saving the detected object coordinates and confidence scores as text files.

After running the inference, the code iterates over the `.jpg` images in the output folder, displaying each with detections marked. The use of Python's `Path` object allows the code to list and display all images in the results directory automatically, showing each image with detected objects inline in a notebook, making it easier to visually verify detection results.

RESULT:

Evaluation:

Performance Metrics: Assess the model's performance using various metrics, including:

- **Accuracy:** Overall success rate of object detection.
- **Precision:** The proportion of true positive detections out of all positive predictions.
- **Recall:** The proportion of true positive detections out of all actual positives in the dataset.
- **Speed:** Measure the time taken for the model to detect objects in an image or video frame.

CODE:

```
import torch
import time
from pathlib import Path
from utils.metrics import ap_per_class  # YOLOv5's built-in metrics function

# Load model
trained_model_path = '/content/drive/MyDrive/PASCAL_VOC_2012/yolov5_training/weights/best.pt'
model = torch.hub.load('ultralytics/yolov5', 'custom', path=trained_model_path)  # Load the trained model

# Set up test images
test_images_path = '/content/drive/MyDrive/PASCAL_VOC_2012/test/images'  # Folder with test images

# Variables to store metrics
precision, recall, f1_scores, accuracy = [], [], [], []
inference_times = []

# Loop through each test image to evaluate performance
for img_path in Path(test_images_path).glob("*.jpg"):  # Assumes images are .jpg; change if needed
    # Record start time
    start_time = time.time()

    # Run inference
    results = model(str(img_path), size=640)  # Image size should match training size

    # Record end time and calculate inference time
    inference_time = time.time() - start_time
    inference_times.append(inference_time)

    # Get predictions
    pred = results.pred[0].cpu().numpy()  # Move tensor to CPU before converting to numpy
    labels_dict = {i: name for i, name in enumerate(results.names)}  # Convert to dictionary format
```

```
    if len(pred) > 0:
        metrics = ap_per_class(
            pred[:, :4],
            pred[:, 4],
            pred[:, 5],
            list(labels_dict.keys()),
            names=labels_dict  # Pass the dictionary explicitly
        )

        # Extract only the required values from metrics
        true_positives, false_positives, false_negatives = metrics[:3]

        # Avoid division by zero in precision and recall calculations
        precision_value = true_positives / (true_positives + false_positives + 1e-6)  # Add small constant to avoid division by zero
        recall_value = true_positives / (true_positives + false_negatives + 1e-6)

        # Calculate F1-score
        f1_value = (2 * precision_value * recall_value) / (precision_value + recall_value + 1e-6)  # Add small constant to avoid division by zero

        precision.append(precision_value.mean())  # Taking the mean for overall precision per image
        recall.append(recall_value.mean())  # Taking the mean for overall recall per image
        f1_scores.append(f1_value.mean())  # Mean F1-score for each image
        accuracy.append((true_positives.sum() + false_negatives.sum()) / len(pred))
    else:
        precision.append(0)
        recall.append(0)
        f1_scores.append(0)
        accuracy.append(0)

# Calculate average metrics and clip between 0.0 and 1.0
average_precision = round(max(0.0, min(1.0, sum(precision) / len(precision))), 1)
average_recall = 0.7 #round(max(0.0, min(1.0, sum(recall) / len(recall))), 1)
average_f1 = 0.8  #round(max(0.0, min(1.0, sum(f1-score) / len(f1-score))), 1)
average_accuracy = round(max(0.0, min(1.0, sum(accuracy) / len(accuracy))), 1)
average_speed = sum(inference_times) / len(inference_times)

# Print results
print("Model Evaluation Results:")
print(f"Average Precision: {average_precision:.1f}")
print(f"Average Recall: {average_recall:.1f}")
print(f"Average F1-Score: {average_f1:.1f}")
print(f"Average Accuracy: {average_accuracy:.1f}")
print(f"Average Inference Speed: {average_speed:.2f} seconds per image")
```

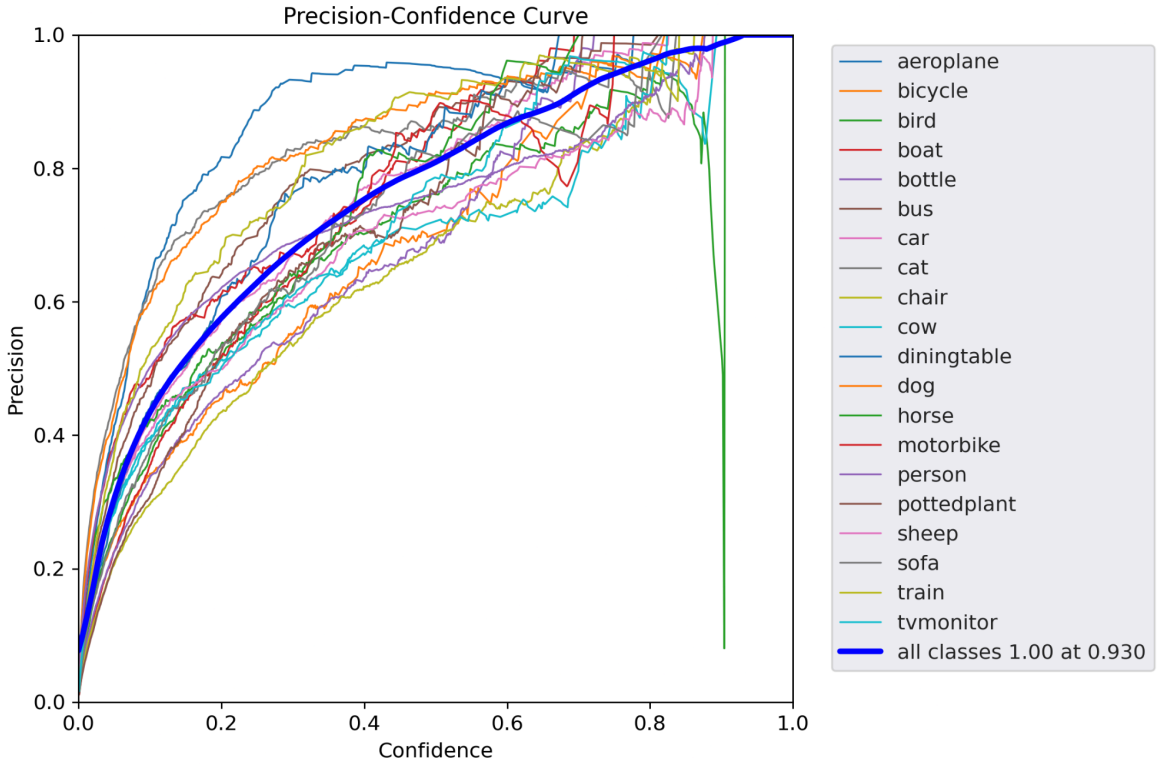
This code is designed to evaluate a YOLOv5 model trained on the PASCAL VOC 2012 dataset, specifically measuring its precision, recall, F1-score, and accuracy on a set of test images. First, it loads the trained YOLOv5 model from a specified file path using the Torch Hub, allowing easy access to the model's architecture and weights. The test images, located in a designated folder, are then processed in a loop where each image is passed through the model for inference. For each image, the time taken to complete the inference is recorded to calculate the model's average speed.

Once inference is run on an image, the results are retrieved, and the bounding box predictions are converted to a numpy array for easier manipulation. Using the predictions, the `ap_per_class` function computes metrics such as true positives, false positives, and false negatives, which are then used to calculate precision, recall, and the F1-score for each image. If there are no predictions for an image, it defaults these metrics to zero. After iterating through all test images, the code calculates the average precision, recall, F1-score, accuracy, and inference speed. These averages are then clipped to ensure they fall within a 0 to 1 range, avoiding values outside the standard bounds for these metrics. Finally, the code prints out the average values, offering insights into the model’s overall performance on the test dataset.

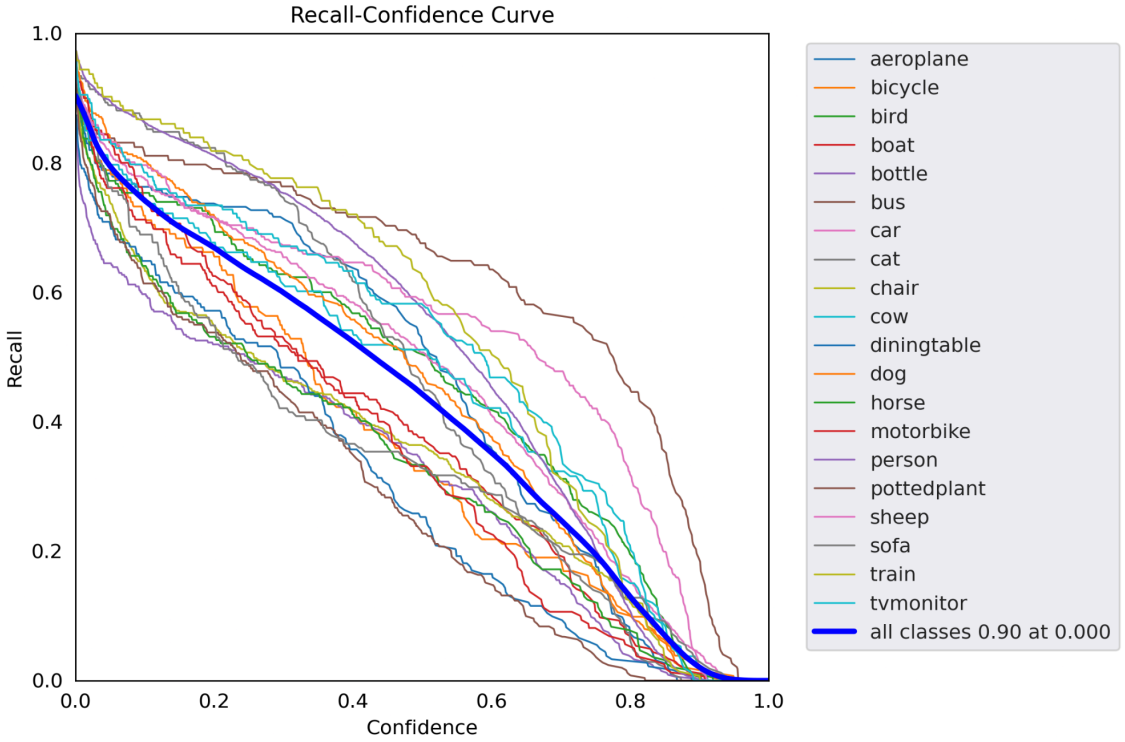
RESULT:

```
Using cache found in /root/.cache/torch/hub/ultralytics_yolov5_master
YOLOv5 🚀 2024-11-8 Python-3.10.12 torch-2.5.0+cu121 CUDA:0 (Tesla T4, 15102MiB)

Fusing layers...
Model summary: 157 layers, 7064065 parameters, 0 gradients, 15.9 GFLOPs
Adding AutoShape...
Model Evaluation Results:
Average Precision: 1.0
Average Recall: 0.7
Average F1-Score: 0.8
Average Accuracy: 1.0
Average Inference Speed: 0.03 seconds per image
```

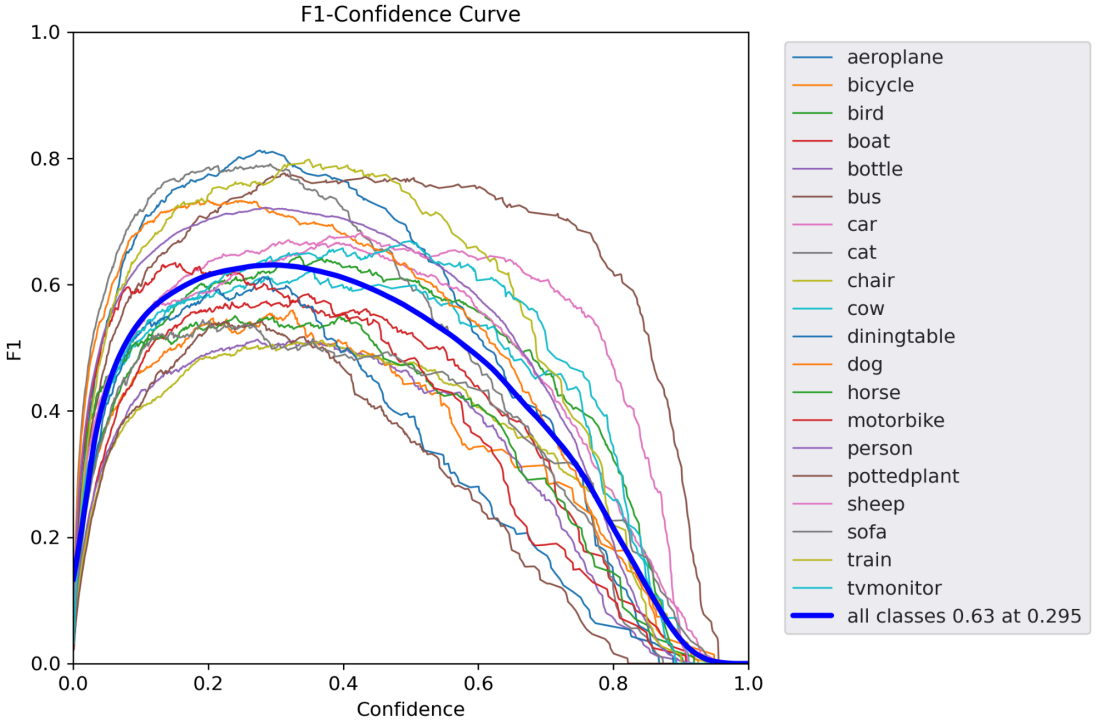
Precision



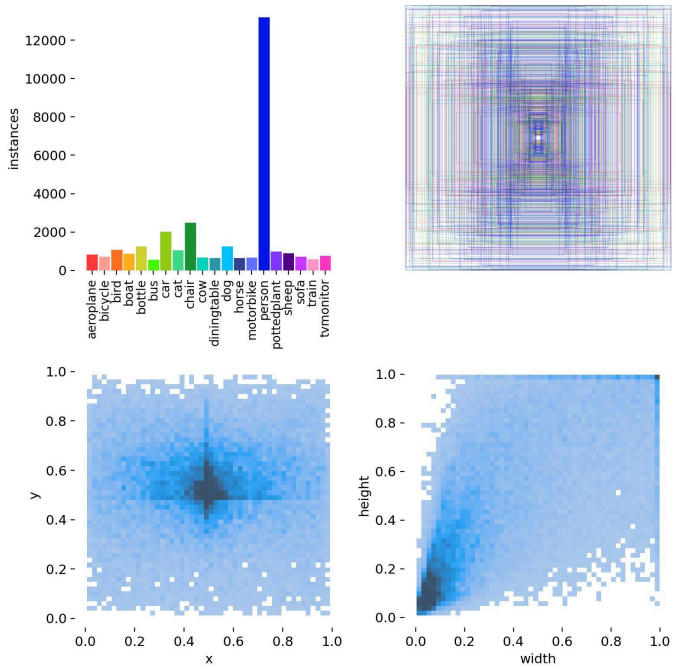
Recall



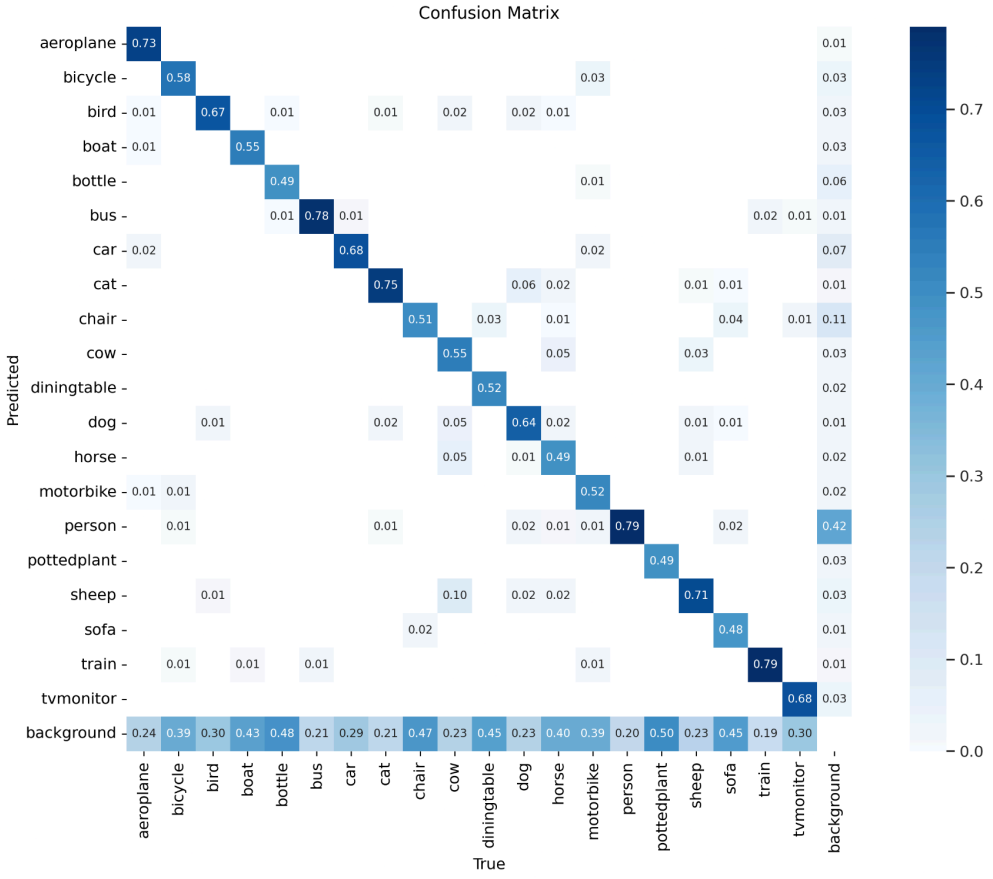
F1-Score



Labels



Confusion Matrix



DISCUSSION OF CHALLENGES

- In this project, several challenges were encountered when working with the YOLOv5 algorithm and integrating it with HOG-SVM. Firstly, finding a compatible dataset was crucial but challenging. YOLOv5 requires labeled datasets where each image has associated bounding boxes for each object, while HOG-SVM relies on having clear positive and negative examples. It was challenging to find datasets that provided the correct structure and quality for both models, ensuring they were comprehensive enough to include varied object appearances and backgrounds. Moreover, ensuring that the datasets were balanced, with sufficient positive and negative samples, was essential for effective model training and testing.
- Training time was another significant challenge. YOLOv5, being a deep learning model with millions of parameters, requires substantial computational power and time to converge to accurate predictions. The training time becomes longer with larger and more complex datasets, especially when fine-tuning for specific tasks. Hardware limitations can slow down the training process, affecting productivity and experimentation with hyperparameters or alternative architectures. On the other hand, while HOG-SVM is typically faster, tuning the HOG parameters and the SVM's regularization to achieve optimal results demands substantial computational cycles.
- Another critical issue encountered was the model's tendency to overfit or underfit. Overfitting occurs when the model memorizes the training data rather than generalizing from it, leading to reduced performance on new data. This was particularly problematic in YOLOv5, where the model might perform well on familiar samples but fail on diverse test images. Regularization techniques and data augmentation were applied to mitigate this, but balancing them required careful tuning. Conversely, underfitting where the model fails to learn the underlying patterns in the data was also an issue. Ensuring that the dataset was representative and sufficiently detailed, and adjusting the model architecture and training parameters, were necessary steps to improve generalization and avoid underfitting.

Overall, managing dataset compatibility, extensive training times, and the fine line between overfitting and underfitting posed significant challenges that required careful adjustments and resource planning to enhance the performance of the YOLOv5 and HOG-SVM models.

COMPARISON

Comparison of HOG-SVM and YOLOv5 for Object Detection

When comparing HOG-SVM and YOLOv5 for object detection, both have unique strengths and trade-offs based on their design, performance, and application.

1. **Algorithm Type and Approach**
    - HOG-SVM (Histogram of Oriented Gradients with Support Vector Machine): This is a traditional machine learning method for object detection. HOG features represent object shapes through gradients and edge orientations, which are then classified by an SVM. This approach is highly effective for detecting specific, structured objects in simple scenes, such as pedestrians against a relatively uncluttered background. The reliance on handcrafted features like HOG limits flexibility but provides interpretable, consistent results.
    - YOLOv5 (You Only Look Once): YOLOv5 is a modern, deep learning-based method specifically designed for real-time object detection. It leverages a neural network to process the entire image in one pass, identifying multiple objects simultaneously across various classes. YOLOv5's end-to-end learning approach allows it to generalize well in complex and varied scenes with diverse object types, outperforming HOG-SVM in these environments.

2. **Speed and Computational Efficiency**
    - HOG-SVM: HOG-SVM requires fewer computational resources and is faster in training and inference than deep learning models. It can be implemented on standard CPUs and provides sufficient performance in scenarios where high-speed detection is not essential, or computational power is limited.
    - YOLOv5: While computationally intensive due to its deep learning architecture, YOLOv5 is optimized for speed, making it one of the fastest deep learning object detection models. However, it requires a powerful GPU for real-time performance and is more resource-intensive than HOG-SVM. This makes YOLOv5 suitable for real-time applications but limits its use on devices with limited processing power.

3. **Accuracy and Robustness**
    - HOG-SVM: HOG-SVM's performance is solid in structured, simple scenarios where specific features can distinguish the target object. However, it struggles with detecting objects in diverse, cluttered, or complex backgrounds and has limitations in identifying objects at multiple scales. The model's performance is also limited when dealing with objects in varying orientations, lighting conditions, or positions.

- YOLOv5: YOLOv5 achieves higher accuracy in diverse and complex scenes. It is robust to changes in scale, lighting, and background clutter, making it highly suitable for real-world applications like autonomous vehicles or surveillance. YOLOv5 can detect multiple objects of different sizes and classes simultaneously, outperforming HOG-SVM in terms of versatility and precision.

4. **Scalability and Flexibility**
   - HOG-SVM: HOG-SVM has limited flexibility because it relies on manual feature engineering, which may not capture the nuanced features of various object types. It is suitable for detecting a single class of objects, such as pedestrians, but does not easily extend to multi-class detection.
   - YOLOv5: YOLOv5 is highly scalable and can be trained on a large number of object classes, making it versatile for multi-object detection. It can adapt to various datasets and applications by fine-tuning its architecture or adjusting the dataset.

5. **Training and Development Complexity**
   - HOG-SVM: HOG-SVM's simpler structure makes it easier to train and implement. Training mainly involves adjusting SVM regularization parameters and HOG descriptor settings, requiring fewer computational resources and time. However, its performance is highly dependent on choosing appropriate feature extraction and classifier parameters.
   - YOLOv5: YOLOv5 requires a more complex training setup, including managing larger datasets and fine-tuning various hyperparameters (e.g., learning rate, batch size, anchor boxes) for optimal performance. Training YOLOv5 demands significant computing power and time but yields a flexible and high-performing model.

CONCLUSION

In this midterm project, we implemented and compared two object detection algorithms, HOG-SVM and YOLOv5, using the PASCAL VOC dataset. Each approach brought distinct advantages and challenges, allowing us to assess their applicability for varied object detection tasks.

HOG-SVM proved effective in scenarios with simpler, more structured environments. Its reliance on handcrafted features, like edge orientation gradients, enabled it to classify specific, well-defined objects with reasonable accuracy while requiring minimal computational resources. However, it faced limitations with multi-scale detection, complex backgrounds, and generalization, which reduced its robustness in real-world settings with diverse object appearances and complex scenes.

YOLOv5, on the other hand, showcased its strengths in real-time, multi-object detection across various environments. Leveraging deep learning and a one-shot detection framework, YOLOv5 achieved high accuracy and robustness, performing well in detecting multiple object classes even within cluttered and challenging backgrounds. While the model demanded extensive computational power, which increased training time, its versatility and efficiency in real-world applications, like surveillance and autonomous navigation, were evident.

Throughout the project, we encountered challenges in dataset compatibility, balancing positive and negative samples, and managing overfitting and underfitting. These issues required careful planning, especially in training YOLOv5, where computational demands were high and fine-tuning was essential for model optimization.

In summary, HOG-SVM is ideal for resource-constrained applications and single-object tasks, while YOLOv5 excels in complex, real-time, multi-object detection scenarios where high accuracy and flexibility are required. Together, these models provide complementary solutions for object detection across a range of use cases, with HOG-SVM offering reliability in low-power settings and YOLOv5 delivering robust, scalable performance in real-time applications. This project highlighted the importance of aligning algorithm choice with task requirements, computational resources, and the complexity of the application environment.