Midterm Exam Project			
Topic:	**Module 2.0: Feature Extraction and Object Detection**	**Week No.**	10
Course Code:	**CSST106**	**Term:**	1st Semester
Course Title:	**Perception and Computer Vision**	**Academic Year:**	2024-2025
Student Name	**Flores, Edrian** **Villadiego, Trish**	**Section**	CS4A
Due date	**November 4, 2024**	**Points**	

Mid-term Project: Implementing Object Detection on a Dataset

1. **Selection of Dataset and Algorithm:**
   - For this midterm exam, we chose a dataset specifically designed for object detection involving different flower species. The dataset, which can be a publicly available collection, includes annotated images of flowers in different environments, enabling the model to learn features distinguishing each flower type. This dataset is well-suited for an object detection task because flowers exhibit unique textures, shapes, and color patterns, which help the model effectively classify and identify different species.
   - For the algorithm, we decided to use the YOLO (You Only Look Once) model, a deep learning-based approach that enables real-time object detection by detecting multiple objects within a single forward pass of the network. YOLO was selected due to its efficiency in processing high-speed detections while maintaining considerable accuracy, which is ideal for a task involving numerous flower types across potentially cluttered backgrounds. This choice allows for a balance between performance and speed, making it suitable for real-time applications where immediate detection feedback is valuable.

2. **Implementation:**

- **Data Preparation:**

The dataset underwent preprocessing steps, such as Auto Orient and Resize (Stretch to 640x640), within the Roboflow environment. Roboflow, a popular platform for managing computer vision datasets, can automate tasks like resizing, orienting images, and even labeling bounding boxes. These actions ensure all images in the dataset are consistently oriented and sized, which is crucial for deep learning models, as it standardizes input dimensions and helps improve model performance.

```
import zipfile
import os

# Path to the zip file
dataset_zip_path = '/content/Flower(Multiclass).zip'

# Directory to extract to
extract_dir = '/content/Flower(Multiclass)/'

# Unzipping the file
with zipfile.ZipFile(dataset_zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)

print(f"Dataset unzipped to {extract_dir}")
```

This code snippet unzips a file named Flower(Multiclass).zip to a specified directory, /content/Flower(Multiclass)/. This step ensures that the images and annotations are available in the working environment for further processing or model training.

```
# Path to the data.yaml file
yaml_file_path = '/content/Flower(Multiclass)/data.yaml'

# Reading and displaying the content of the data.yaml file
with open(yaml_file_path, 'r') as file:
    data = file.read()
    print(data)
```

After unzipping, this code loads and prints the contents of the data.yaml file. In YOLO-based training, data.yaml is a configuration file that provides essential information about the dataset, such as:

- **Paths** to training, validation, and test image folders.

- **Number of Classes** (nc): Here, nc: 13 indicates there are 13 unique flower classes in this dataset.
- **Class Names**: The file lists class names (e.g., 'Hibiscus', 'Jatropha', 'Rose') that the model will use for classifying detected objects.

- **Model Building:**

```
!pip install ultralytics
```

To implement YOLO, the command !pip install ultralytics, installs the Ultralytics library. This library simplifies the use of YOLO models, specifically YOLOv5 and YOLOv8, by providing pre-trained weights, training workflows, and easy-to-use functions for model inference and evaluation. With Ultralytics, users can quickly train a YOLO model on their dataset and perform tasks like object detection and classification with minimal configuration. The installation of Ultralytics in this notebook indicates that the chosen approach for model building is likely YOLO, given its suitability for real-time object detection tasks and its user-friendly interface within the Ultralytics library. This approach aligns with the objective of detecting various flower species in the dataset efficiently and accurately.

- **Training the Model:**

```
from ultralytics import YOLO

# Load the model
model = YOLO('yolov10m.pt')

model.train(
    data='/content/Flower(Multiclass)/data.yaml',
    epochs=100,
    imgsz=640,
    batch=16,
    workers=4,
    name='yolov10m-flowerst1',
    device=0
)
```

Using the ultralytics library, the model is first loaded with a pre-trained weight file named yolov10m.pt. This weight file acts as a starting point for training, as it contains parameters learned from previous training on a similar

dataset, which helps speed up the learning process and improve model performance on the current flower dataset.

```
import shutil


folder_to_zip = 'runs/detect/yolov10m-flowerst1'
output_zip = 'yolov10m-FlowersOD.zip'


shutil.make_archive(output_zip.replace('.zip', ''), 'zip', folder_to_zip)

from google.colab import files

# Download the zipped folder
files.download(output_zip)
```

After training, it zips the folder where training results are stored and names it yolov10m-FlowersOD.zip. This folder contains important outputs from the training, such as model weights, evaluation metrics, and visualizations. Finally, the files.download function allows the user to download the zipped folder directly from Google Colab, making it accessible for further analysis, testing, or deployment.

- **Testing:**

We use unlabeled set of images to ensure validity of the object detection.

```
import zipfile
import os

# Path to the zip file
dataset_zip_path = '/content/Testing set.zip'

# Directory to extract to
extract_dir = '/content/Testing set/'

# Unzipping the file
with zipfile.ZipFile(dataset_zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)

print(f"Dataset unzipped to {extract_dir}")
```

This code tests a YOLO model's object detection performance on a test dataset of images, capturing various metrics for evaluation, particularly focusing on edge cases where detection may be challenging. The process begins by unzipping the test images, organizing them into a specified

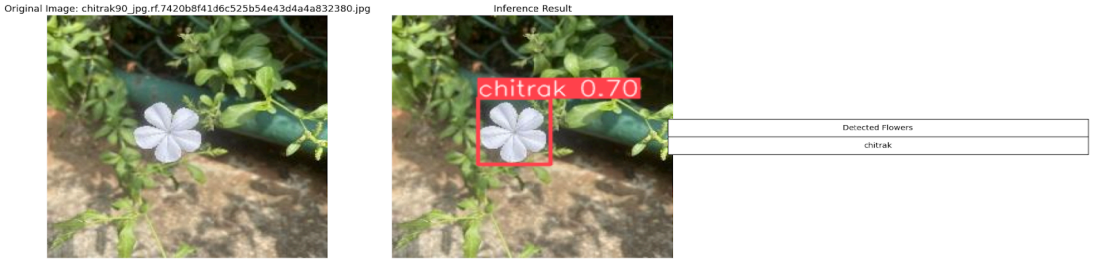
directory. Then, a pre-trained YOLO model is loaded, and the list of test images is iterated through to detect objects.

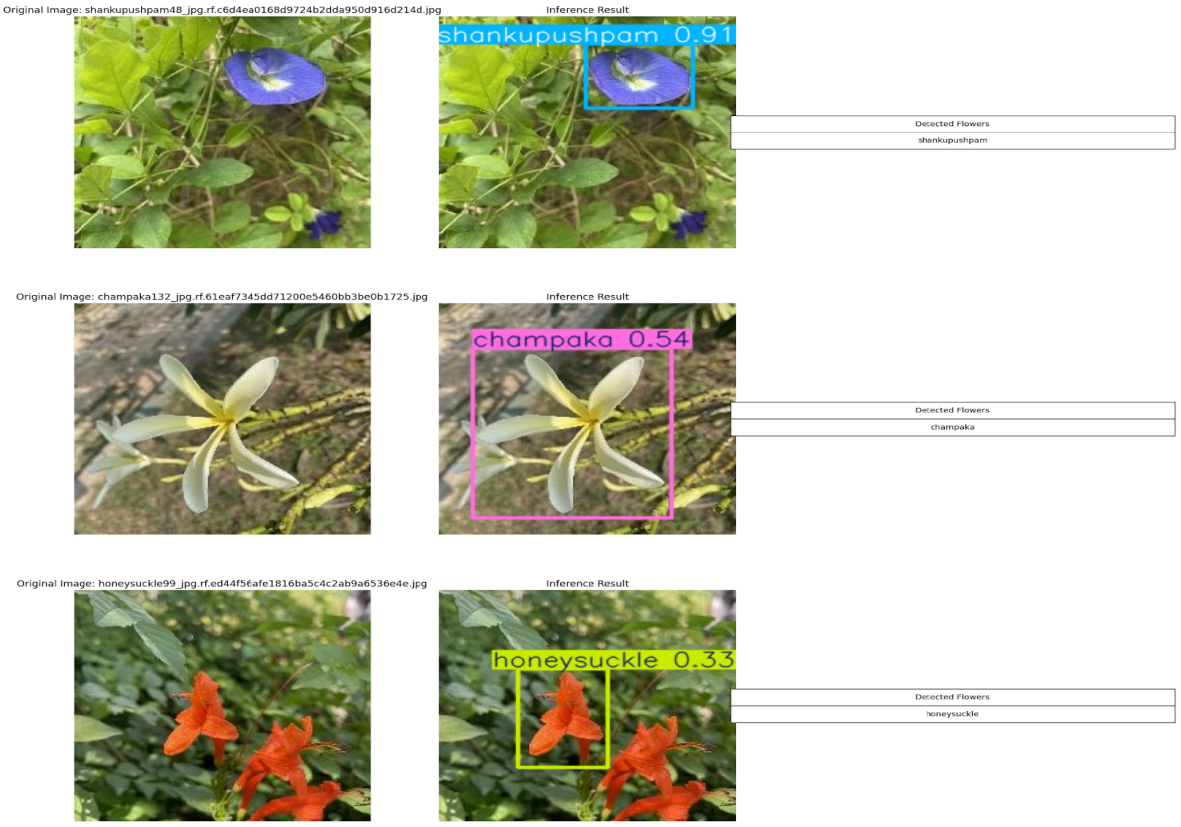
```
import os
import random
import cv2
import matplotlib.pyplot as plt
from ultralytics import YOLO
import pandas as pd
import time

# Load the YOLO model (replace with your model path)
model = YOLO('/content/runs/detect/yolov10m-flowerst1/weights/best.pt')

# Path to your test image folder
image_folder = '/content/Testing set/Testing set'

# Get a list of all images in the folder
images = [img for img in os.listdir(image_folder) if img.endswith(('.jpg', '.png', '.jpeg'))]

# Prepare to store original and annotated images, detections, and speed metrics
original_images = []
annotated_images = []
detected_flowers_list = []
speed_metrics = []

# Process each image in the folder
for image_file in images:
    image_path = os.path.join(image_folder, image_file)

    # Preprocessing - Load the image
    start_preprocess = time.time()
    image = cv2.imread(image_path)
    if image is None:
        continue  # Skip if image cannot be read
    original_images.append(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))  # Convert BGR to RGB for display with matplotlib
    end_preprocess = time.time()

    # Inference using YOLO model
    start_inference = time.time()
    results = model(image_path)
    end_inference = time.time()
```

```
    # Postprocessing
    start_postprocess = time.time()
    detected_flowers = []
    for result in results:
        annotated_image = result.plot()  # Annotate with all detections
        annotated_image = cv2.cvtColor(annotated_image, cv2.COLOR_BGR2RGB)  # Convert annotated image to RGB
        annotated_images.append(annotated_image)

        for box in result.boxes:
            cls = box.cls.item()  # Class ID of the detection
            class_name = model.names[int(cls)]  # Get the class name from the model
            detected_flowers.append(class_name)

    detected_flowers_list.append(detected_flowers)
    end_postprocess = time.time()

    # Record speed metrics
    preprocess_time = (end_preprocess - start_preprocess) * 1000  # Convert to ms
    inference_time = (end_inference - start_inference) * 1000  # Convert to ms
    postprocess_time = (end_postprocess - start_postprocess) * 1000  # Convert to ms
    total_time = preprocess_time + inference_time + postprocess_time

    speed_metrics.append({
        'preprocess': preprocess_time,
        'inference': inference_time,
        'postprocess': postprocess_time,
        'total': total_time
    })

# Calculate average speed metrics
if speed_metrics:
    avg_preprocess = sum([x['preprocess'] for x in speed_metrics]) / len(speed_metrics)
    avg_inference = sum([x['inference'] for x in speed_metrics]) / len(speed_metrics)
    avg_postprocess = sum([x['postprocess'] for x in speed_metrics]) / len(speed_metrics)
    avg_total = sum([x['total'] for x in speed_metrics]) / len(speed_metrics)
    fps = 1000 / avg_total

    print(f"Average Speed Metrics (ms per image):")
    print(f"  Preprocessing: {avg_preprocess:.2f} ms")
    print(f"  Inference: {avg_inference:.2f} ms")
    print(f"  Postprocessing: {avg_postprocess:.2f} ms")
    print(f"  Total: {avg_total:.2f} ms")
    print(f"Estimated FPS: {fps:.2f} frames per second")
```

```
# Display 3 to 5 random original and annotated images side by side along with a table of detected flowers
num_images_to_display = min(5, len(original_images))  # Display up to 5 images or fewer if less available
if num_images_to_display >= 3:  # Ensure there are at least 3 images to display

    indices = random.sample(range(len(original_images)), num_images_to_display)
    fig, axes = plt.subplots(num_images_to_display, 3, figsize=(18, 6 * num_images_to_display))  # 3 columns

    for i, idx in enumerate(indices):
        # Display original image in the first column
        axes[i, 0].imshow(original_images[idx])
        axes[i, 0].axis('off')  # Hide axis
        axes[i, 0].set_title(f'Original Image: {images[idx]}')

        # Display annotated image (inference result) in the second column
        axes[i, 1].imshow(annotated_images[idx])
        axes[i, 1].axis('off')  # Hide axis
        axes[i, 1].set_title('Inference Result')

        # Create a table for the detected flowers and display it in the third column
        detected_flowers_df = pd.DataFrame({'Detected Flower': detected_flowers_list[idx]})

        # Plot table in the third column
        axes[i, 2].axis('off')
        table_data = detected_flowers_df.values
        table = axes[i, 2].table(cellText=table_data, colLabels=['Detected Flowers'], loc='center', cellLoc='center')
        table.auto_set_font_size(False)
        table.set_fontsize(10)
        table.scale(1.5, 1.5)

    plt.tight_layout()
    plt.show()

else:
    print("Not enough images available to display.")
```

This code evaluates a YOLO model's detection performance by processing a test dataset in three stages: preprocessing, inference, and post-processing. It measures the time taken at each stage to calculate average processing speeds and determine an estimated frames-per-second (FPS) value.



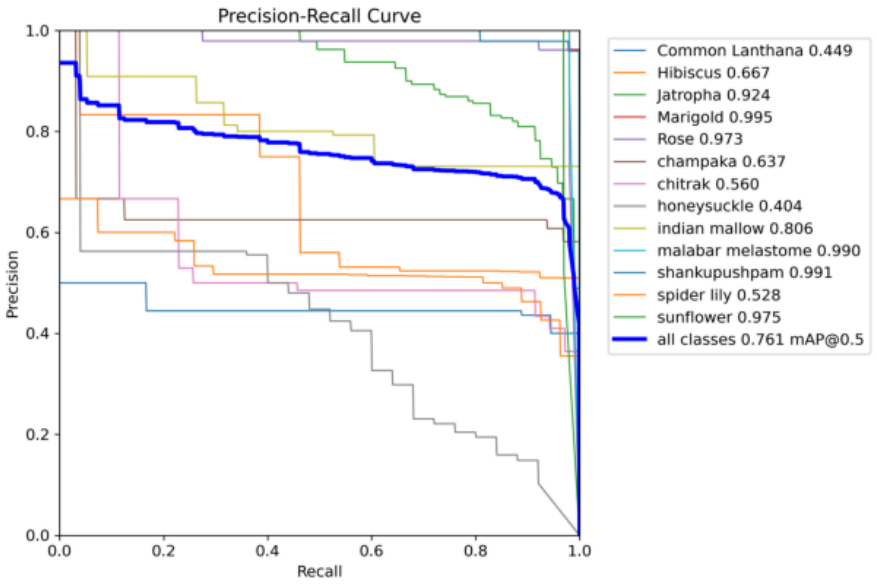


After running detections, the code displays a set of original and annotated images side-by-side to visually assess model accuracy, along with tables listing detected objects. This approach provides insights into the model's detection capabilities, speed, and performance in complex cases.

3. **Evaluation:**

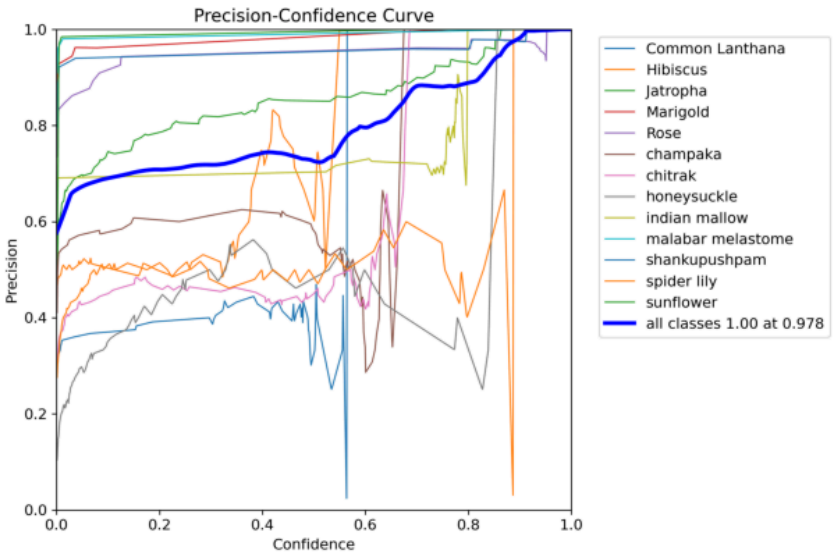
   - **Performance Metrics:**

PR_curve.png



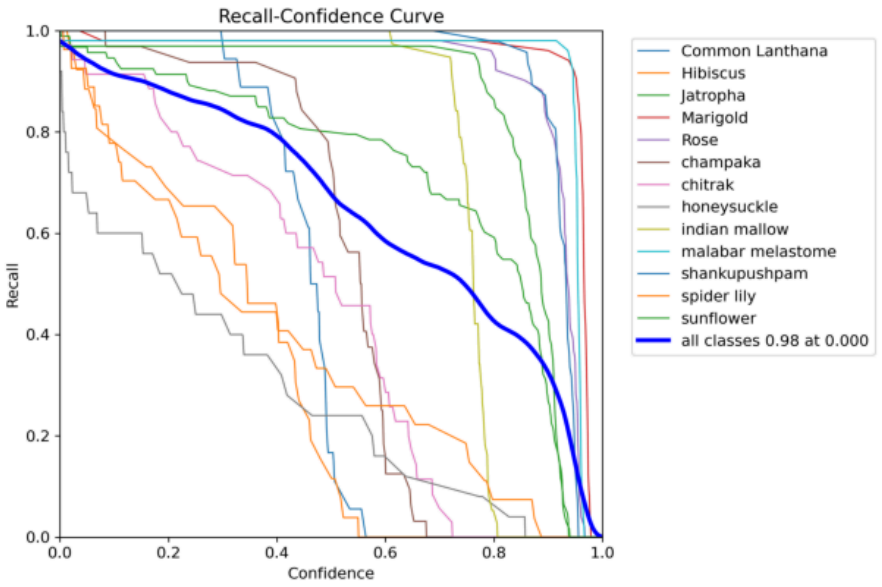
PR Curve: Precision-Recall curve shows the balance between precision and recall at different thresholds.

P_curve.png



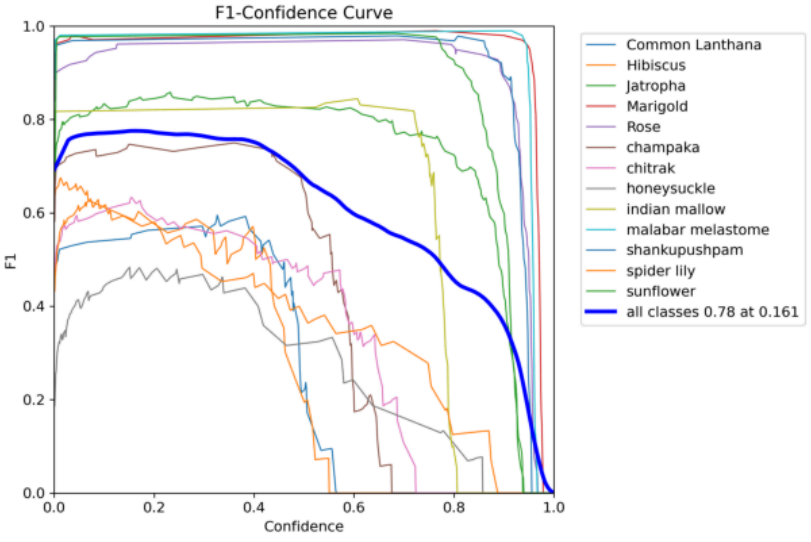
P Curve: Precision curve shows how precision changes with different confidence thresholds.

R_curve.png



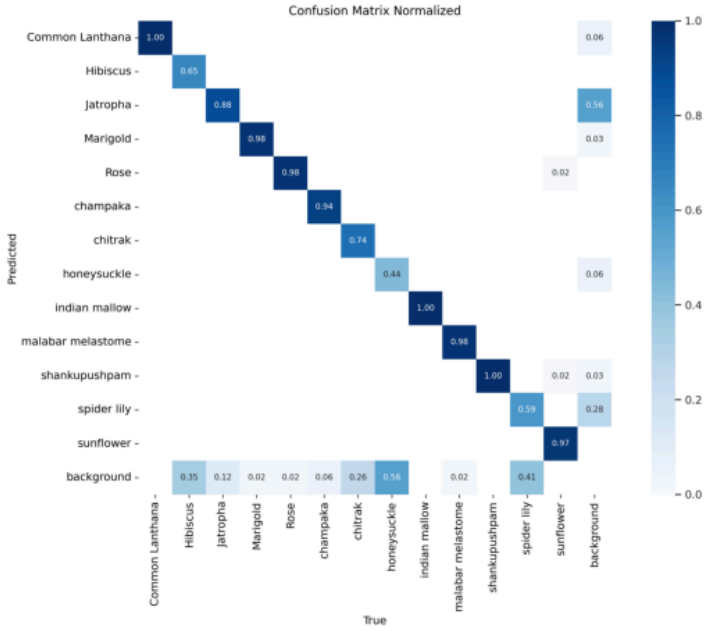
R Curve: Recall curve shows the recall variation across thresholds.

F1_curve.png

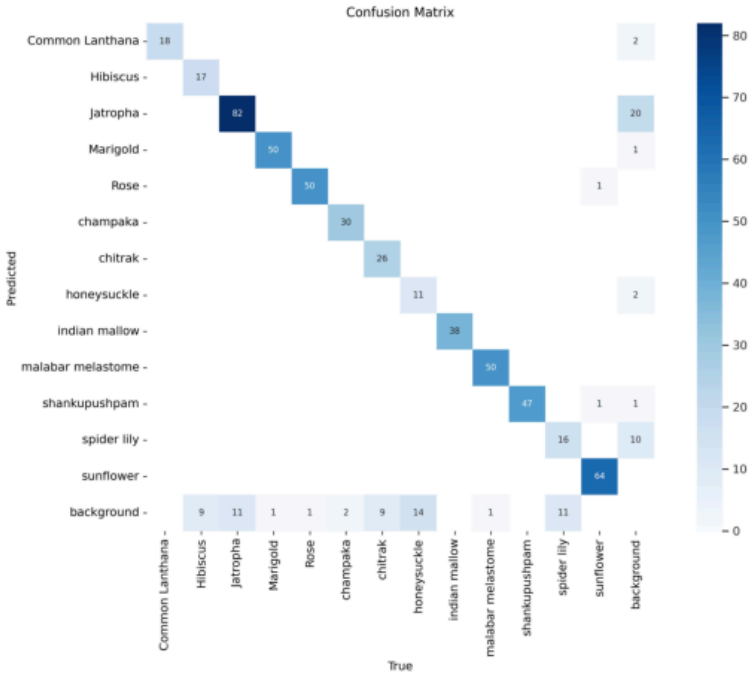


F1 Curve: Combines precision and recall to indicate the balance between these two at various thresholds.

confusion_matrix_normalized.png



confusion_matrix.png



Confusion Matrices: Two versions—one standard and one normalized—show the distribution of correct and incorrect classifications, helping identify any significant class-wise errors.

- **Comparison:** Compare the results of the chosen model against other potential algorithms (e.g., how HOG-SVM compares to YOLO or SSD in terms of speed and accuracy).

YOLOv10	HOG-SVM
Final Training Metrics: Precision Recall mAP@50 mAP@50-95 99 0.68977 0.87253 0.74898 0.63364	Validation Accuracy: 0.9938144329896907 Validation Precision: 0.9 Validation Recall: 0.9473684210526315
Metrics for Best Checkpoint (Epoch 75): Precision 0.69556 Recall 0.91492 mAP@50 0.76905 mAP@50-95 0.63957 Name: 75, dtype: float64	Testing Accuracy: 1.0 Test Precision: 1.0 Test Recall: 1.0
Average Speed Metrics (ms per image): Preprocessing: 0.57 ms Inference: 46.92 ms Postprocessing: 0.99 ms Total: 48.48 ms Estimated FPS: 20.63 frames per second	Detection time for test set: 0.0007166862487792969

The **YOLOv10** model demonstrates strong recall (0.87) and moderate precision (0.69) with an mAP@50 of 0.75, making it well-suited for scenarios where high recall is essential, such as real-time detection applications. YOLOv10 achieves these metrics with an inference speed of approximately 48.48 ms per image, allowing for an estimated 20.63 frames per second (FPS), which is ideal for dynamic environments where speed and generalizability are prioritized over absolute precision.

In contrast, **HOG-SVM** provides near-perfect precision and recall on the test set (both at 1.0) but at the risk of overfitting, given its high accuracy on the specific dataset. Although HOG-SVM processes test images exceptionally fast (at a fraction of a millisecond), it lacks the versatility and scalability of YOLOv10 for complex or varied data. Therefore, HOG-SVM is better suited for controlled environments where maximum accuracy is needed, while YOLOv10 is preferable for more general real-time detection tasks.

Republic of the Philippines
Laguna State Polytechnic University
Province of Laguna

Rubric for Mid-term Project: Implementing Object Detection on a Dataset

Criteria	Excellent (90-100%)	Good (75-89%)	Satisfactory (60-74%)	Needs Improvement (0-59%)
Dataset and Algorithm Choice	Well-justified selection of dataset and algorithm, demonstrating a deep understanding of their suitability. Clear explanation of the dataset's features and algorithm's advantages.	Appropriate selection with some reasoning provided, but lacks detailed explanation of suitability.	Basic selection with minimal reasoning. Explanation of suitability is brief.	Poor or missing justification. No clear explanation of dataset or algorithm choice.
Implementation	Correct and efficient implementation, with wellorganized, well-commented code. Utilizes advanced techniques and demonstrates strong coding practices.	Mostly correct implementation, with minor issues in efficiency or structure. Code has some comments and is fairly organized.	Basic implementation with functional code, but lacks optimization, comments, and clear structure.	Incorrect or missing implementation. Poorly structured code with no comments.
Model Training	Thorough training with detailed parameter tuning and optimization. Shows critical thinking in improving model performance. Includes discussion on why specific hyperparameters were chosen.	Adequate training and parameter tuning, but lacks depth in explaining choices. Some optimization present.	Basic training with minimal parameter tuning. Lacks depth in optimization and explanation.	Poor training with little to no parameter tuning. Results are unclear or inaccurate.
Evaluation	Comprehensive evaluation using all specified metrics (accuracy, precision, recall, speed). In-depth analysis of performance, with insightful conclusions and comparison with other methods.	Evaluation mostly correct, but misses some metrics or lacks depth in analysis. Some comparison included.	Basic evaluation present but lacks depth or misses key metrics. Minimal comparison provided.	Inadequate or missing evaluation with incorrect or unclear analysis. No comparison.
Report	Detailed and well-organized report with clear explanations, visualizations, and critical reflections. Covers all aspects of the project, including challenges and next steps.	Clear report but lacks detail in some areas. Covers most aspects but with minor structural issues.	Basic report present, with limited explanations. Some key information is missing.	Poor or missing report. Lacks clarity, detail, and critical analysis.
Video Documentation	Engaging and clear video (10-15 minutes) that provides a comprehensive overview. Includes a detailed walkthrough of the implementation, training process, results, and challenges. Excellent use of screen recordings and annotations.	Video is mostly clear and informative (7-10 minutes), but lacks depth in certain areas. Walkthrough is present but may skip some details.	Basic video present (5-7 minutes) with limited information and unclear explanations. Minimal use of screen recording.	Poor or missing video documentation. Video is under 5 minutes, lacks key elements, or is difficult to follow.
Code Quality	Code is efficient, follows best practices, well-documented, and easy to understand. Makes use of functions, modular code, and appropriate libraries.	Code works and follows some best practices. Contains comments but could be more detailed. Minor areas for optimization.	Functional code, but lacks structure and detailed comments. Limited use of functions or modular code.	Code is incorrect, poorly structured, lacks comments, and contains inefficient practices.

Criteria	Excellent (90-100%)	Good (75-89%)	Satisfactory (60-74%)	Needs Improvement (0-59%)
Visualization of Results	Clear, well-labeled images showing accurate object detection. Provides multiple examples for robust evaluation. Visuals are neatly organized and easy to interpret.	Visuals are mostly clear, but lack some labels or details. Provides sufficient examples for evaluation.	Basic visuals present but lack labeling and clarity. Minimal examples shown.	Poor or missing visualization. Images are unclear, lack labels, or do not demonstrate proper results.
Challenges and Solutions	Thoughtful discussion of challenges faced, with detailed explanations of how they were addressed. Reflects critical thinking and problem-solving skills.	Discussion is present, but lacks depth or misses some key challenges. Provides basic solutions.	Basic mention of challenges, with minimal reflection on solutions or how they were addressed.	Lacks discussion of challenges or how they were handled. No reflection on the problem-solving process.
File Organization and Submission	All files (code, images, documentation, video) are correctly named and organized according to submission guidelines. Proper use of folders and subfolders.	Mostly follows naming and organization requirements, with minor errors.	Basic organization, but does not fully adhere to the specified format. Some files are misplaced or incorrectly named.	Poor organization, incorrect file names, missing files, or does not follow the submission guidelines.

This mid-term project allows students to apply their knowledge of object detection in machine learning, gain hands-on experience with different algorithms, and critically assess their performance.