```
>dronekit) (1.0.0)
Requirement already satisfied: lxml in /usr/local/lib/python3.11/dist-
packages (from pymavlink>=2.2.20->dronekit) (5.4.0)
Downloading dronekit-2.9.2-py3-none-any.whl (56 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.9/56.9 kB 4.9 MB/s eta
0:00:00
onotonic-1.6-py2.py3-none-any.whl (8.2 kB)
Downloading pymavlink-2.4.47-cp311-cp311-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (6.7 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.7/6.7 MB 3.8 MB/s eta
0:00:00
onotonic, pymavlink, dronekit
Successfully installed dronekit-2.9.2 monotonic-1.6 pymavlink-2.4.47

# Consolidate all necessary imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split
import os
import zipfile
from google.colab import drive
from PIL import Image
import torchvision.transforms as transforms
import cv2
from ultralytics import YOLO
import time

import collections
import collections.abc
# Fix for MutableMapping compatibility with dronekit in Python 3.10+
try:
    collections.MutableMapping = collections.abc.MutableMapping
except AttributeError:
    # If collections.MutableMapping already exists (e.g., in older
Python),
    # this line is not needed and would cause an AttributeError
itself.
    pass

from dronekit import connect, APIException, VehicleMode,
LocationGlobalRelative


# Mount Google Drive
```

```
drive.mount('/content/drive')

# Define zip paths
zip_path = '/content/drive/MyDrive/DATASETS/archive.zip'
yolo_zip_path = '/content/drive/MyDrive/DATASETS/yolo_plant_data.zip'

# Define extraction paths
extract_path = '/content/PlantDataset'
yolo_extract_path = '/content/yolo_data'

# Flag to control subsequent steps
plant_data_extracted = False
yolo_data_extracted = False

# Extract dataset zip file
if os.path.exists(zip_path):
    if not os.path.exists(extract_path):
        try:
            with zipfile.ZipFile(zip_path, 'r') as zip_ref:
                zip_ref.extractall(extract_path)
            print("Dataset extracted successfully!")
            plant_data_extracted = True
        except Exception as e:
            print(f"Error extracting {zip_path}: {e}")
    else:
        print("Dataset already extracted.")
        plant_data_extracted = True
else:
    print(f"Error: The file {zip_path} was not found. Skipping plant
dataset extraction and dependent steps.")

# Extract YOLO dataset zip file
if os.path.exists(yolo_zip_path):
    if not os.path.exists(yolo_extract_path):
        try:
            with zipfile.ZipFile(yolo_zip_path, 'r') as zip_ref:
                zip_ref.extractall(yolo_extract_path)
            print("YOLO Dataset extracted successfully!")
            yolo_data_extracted = True
        except Exception as e:
            print(f"Error extracting {yolo_zip_path}: {e}")
    else:
        print("YOLO Dataset already extracted.")
        yolo_data_extracted = True
else:
    print(f"Error: The file {yolo_zip_path} was not found. Skipping
YOLO dataset extraction and dependent steps.")

print(f"yolo_data_extracted is: {yolo_data_extracted}")
```

```
# Define the transform for prediction for the PyTorch model
predict_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])

# Define class names for the PyTorch model
# This should ideally be loaded from the dataset used for training
class_names = ['Apple___Apple_scab', 'Apple___Black_rot',
'Apple___Cedar_apple_rust', 'Apple___healthy', 'Blueberry___healthy',
'Cherry_(including_sour)___Powdery_mildew',
'Cherry_(including_sour)___healthy',
'Corn_(maize)___Cercospora_leaf_spot Gray_leaf_spot',
'Corn_(maize)___Common_rust_', 'Corn_(maize)___Northern_Leaf_Blight',
'Corn_(maize)___healthy', 'Grape___Black_rot',
'Grape___Esca_(Black_Measles)',
'Grape___Leaf_blight_(Isariopsis_Leaf_Spot)', 'Grape___healthy',
'Orange___Haunglongbing_(Citrus_greening)', 'Peach___Bacterial_spot',
'Peach___healthy', 'Pepper,_bell___Bacterial_spot',
'Pepper,_bell___healthy', 'Potato___Early_blight',
'Potato___Late_blight', 'Potato___healthy', 'Raspberry___healthy',
'Soybean___healthy', 'Squash___Powdery_mildew',
'Strawberry___Leaf_scorch', 'Strawberry___healthy',
'Tomato___Bacterial_spot', 'Tomato___Early_blight',
'Tomato___Late_blight', 'Tomato___Leaf_Mold',
'Tomato___Septoria_leaf_spot', 'Tomato___Spider_mites Two-
spotted_spider_mite', 'Tomato___Target_Spot',
'Tomato___Tomato_Yellow_Leaf_Curl_Virus',
'Tomato___Tomato_mosaic_virus', 'Tomato___healthy']
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
Dataset already extracted.
YOLO Dataset already extracted.
yolo_data_extracted is: True
```

```
# Define the model architecture (latest version with 512 FC and
dropout)
class PlantClassifier(nn.Module):
    def __init__(self, num_classes):
        super(PlantClassifier, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)

        # Calculate the size of the flattened layer dynamically
        # We'll use a dummy tensor to calculate the size after conv
and pool layers
        self._to_linear = None
        self.convs = nn.Sequential(
```

```
            nn.Conv2d(3, 16, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self._calculate_flatten_size()

        self.fc1 = nn.Linear(self._to_linear, 512)
        self.fc2 = nn.Linear(512, num_classes)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.3)

    def _calculate_flatten_size(self):
        # Pass a dummy tensor through the convolutional and pooling
layers
        x = torch.randn(1, 3, 224, 224) # Assuming input image size is
224x224
        x = self.convs(x)
        self._to_linear = x[0].shape[0] * x[0].shape[1] *
x[0].shape[2]


    def forward(self, x):
        x = self.convs(x)
        x = x.view(-1, self._to_linear) # Use the calculated flattened
size
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.fc2(x)
        return x

# Initialize device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


# Data loading and splitting for PlantClassifier
data_dir = os.path.join(extract_path, 'plantvillage dataset', 'color')
if plant_data_extracted and os.path.exists(data_dir):
    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(20),
        transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
        transforms.ToTensor()
    ])

    full_dataset = datasets.ImageFolder(data_dir, transform=transform)

    # Define num_classes based on the dataset
    num_classes = len(full_dataset.classes)
```

```
    print("Number of output classes:", num_classes)
    print("Classes:", full_dataset.classes)


    train_size = int(0.8 * len(full_dataset))
    val_size = len(full_dataset) - train_size
    train_dataset, val_dataset = random_split(full_dataset,
[train_size, val_size])

    train_loader = DataLoader(train_dataset, batch_size=32,
shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

    print(f"Train size: {len(train_dataset)}; Validation size:
{len(val_dataset)}")

    # Initialize model, criterion, optimizer, and scheduler
    model = PlantClassifier(num_classes=num_classes).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.1)


    # Training loop
    num_epochs = 10 # Number of epochs for PyTorch model training #
Reduced for faster execution
    best_val_acc = 0.0

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_loss = running_loss / total
```

```
        train_acc = 100 * correct / total

        # Validation phase
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for images, labels in val_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item() * images.size(0)
                _, predicted = torch.max(outputs, 1)
                val_total += labels.size(0)
                val_correct += (predicted == labels).sum().item()

        val_loss = val_loss / val_total
        val_acc = 100 * val_correct / val_total

        print(f"Epoch {epoch+1}/{num_epochs} - "
              f"Train Loss: {train_loss:.4f}, Train Acc:
{train_acc:.2f}% - "
              f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%")

        # Save the best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), 'best_model.pth')
            print("□ Best model saved!")

        # Step the learning rate scheduler
        scheduler.step()

    # Save the final trained PyTorch model
    torch.save(model.state_dict(), "plant_model.pth")
    print("Final model saved as plant_model.pth")
else:
    print(f"Error: Data directory not found at {data_dir} or plant
data not extracted. Skipping PyTorch model training and saving.")

Number of output classes: 38
Classes: ['Apple___Apple_scab', 'Apple___Black_rot',
'Apple___Cedar_apple_rust', 'Apple___healthy', 'Blueberry___healthy',
'Cherry_(including_sour)___Powdery_mildew',
'Cherry_(including_sour)___healthy',
'Corn_(maize)___Cercospora_leaf_spot Gray_leaf_spot',
'Corn_(maize)___Common_rust_', 'Corn_(maize)___Northern_Leaf_Blight',
'Corn_(maize)___healthy', 'Grape___Black_rot',
```

```
'Grape___Esca_(Black_Measles)',
'Grape___Leaf_blight_(Isariopsis_Leaf_Spot)', 'Grape___healthy',
'Orange___Haunglongbing_(Citrus_greening)', 'Peach___Bacterial_spot',
'Peach___healthy', 'Pepper,_bell___Bacterial_spot',
'Pepper,_bell___healthy', 'Potato___Early_blight',
'Potato___Late_blight', 'Potato___healthy', 'Raspberry___healthy',
'Soybean___healthy', 'Squash___Powdery_mildew',
'Strawberry___Leaf_scorch', 'Strawberry___healthy',
'Tomato___Bacterial_spot', 'Tomato___Early_blight',
'Tomato___Late_blight', 'Tomato___Leaf_Mold',
'Tomato___Septoria_leaf_spot', 'Tomato___Spider_mites Two-
spotted_spider_mite', 'Tomato___Target_Spot',
'Tomato___Tomato_Yellow_Leaf_Curl_Virus',
'Tomato___Tomato_mosaic_virus', 'Tomato___healthy']
Train size: 43444; Validation size: 10861
Epoch 1/10 - Train Loss: 1.6792, Train Acc: 52.56% - Val Loss: 0.8708,
Val Acc: 73.02%
 Best model saved!
Epoch 2/10 - Train Loss: 0.9973, Train Acc: 69.76% - Val Loss: 0.6654,
Val Acc: 79.38%
 Best model saved!
Epoch 3/10 - Train Loss: 0.8353, Train Acc: 74.67% - Val Loss: 0.7000,
Val Acc: 77.87%
Epoch 4/10 - Train Loss: 0.5769, Train Acc: 81.98% - Val Loss: 0.4064,
Val Acc: 87.15%
 Best model saved!
Epoch 5/10 - Train Loss: 0.5334, Train Acc: 83.40% - Val Loss: 0.3996,
Val Acc: 87.40%
 Best model saved!
Epoch 6/10 - Train Loss: 0.4979, Train Acc: 84.32% - Val Loss: 0.3584,
Val Acc: 88.86%
 Best model saved!
Epoch 7/10 - Train Loss: 0.4744, Train Acc: 84.90% - Val Loss: 0.3504,
Val Acc: 89.16%
 Best model saved!
Epoch 8/10 - Train Loss: 0.4706, Train Acc: 84.90% - Val Loss: 0.3498,
Val Acc: 89.14%
Epoch 9/10 - Train Loss: 0.4653, Train Acc: 85.18% - Val Loss: 0.3493,
Val Acc: 88.97%
Epoch 10/10 - Train Loss: 0.4564, Train Acc: 85.41% - Val Loss:
0.3362, Val Acc: 89.48%
 Best model saved!
Final model saved as plant_model.pth
```

```
# YOLO model training and validation
yolo_data_yaml = os.path.join(yolo_extract_path,
'/content/yolo_data/data.yaml')
if yolo_data_extracted and os.path.exists(yolo_data_yaml):
    yolo_model = YOLO('yolov8n.pt')
    # The number of epochs for YOLO training is set here:
```

```
    results = yolo_model.train(data=yolo_data_yaml, epochs=50,
imgsz=640, batch=16)
    yolo_model.val()
    # Print the path to the best saved model
    if hasattr(results, 'save_dir'):
        best_model_path = os.path.join(results.save_dir, 'weights',
'best.pt')
        print(f"Best YOLO model saved to: {best_model_path}")

else:
    print(f"Error: YOLO /content/yolo_data/data.yaml not found at
{yolo_extract_path}//content/yolo_data/data.yaml or YOLO data not
extracted. Skipping YOLO model training and validation.")
```

```
Downloading
https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8n
.pt to 'yolov8n.pt'...

100%|██████████| 6.25M/6.25M [00:00<00:00, 273MB/s]

Ultralytics 8.3.159 □ Python-3.11.13 torch-2.6.0+cu124 CUDA:0 (Tesla
T4, 15095MiB)
engine/trainer: agnostic_nms=False, amp=True, augment=False,
auto_augment=randaugment, batch=16, bgr=0.0, box=7.5, cache=False,
cfg=None, classes=None, close_mosaic=10, cls=0.5, conf=None,
copy_paste=0.0, copy_paste_mode=flip, cos_lr=False, cutmix=0.0,
data=/content/yolo_data/data.yaml, degrees=0.0, deterministic=True,
device=None, dfl=1.5, dnn=False, dropout=0.0, dynamic=False,
embed=None, epochs=50, erasing=0.4, exist_ok=False, fliplr=0.5,
flipud=0.0, format=torchscript, fraction=1.0, freeze=None, half=False,
hsv_h=0.015, hsv_s=0.7, hsv_v=0.4, imgsz=640, int8=False, iou=0.7,
keras=False, kobj=1.0, line_width=None, lr0=0.01, lrf=0.01,
mask_ratio=4, max_det=300, mixup=0.0, mode=train, model=yolov8n.pt,
momentum=0.937, mosaic=1.0, multi_scale=False, name=train, nbs=64,
nms=False, opset=None, optimize=False, optimizer=auto,
overlap_mask=True, patience=100, perspective=0.0, plots=True,
pose=12.0, pretrained=True, profile=False, project=None, rect=False,
resume=False, retina_masks=False, save=True, save_conf=False,
save_crop=False, save_dir=runs/detect/train, save_frames=False,
save_json=False, save_period=-1, save_txt=False, scale=0.5, seed=0,
shear=0.0, show=False, show_boxes=True, show_conf=True,
show_labels=True, simplify=True, single_cls=False, source=None,
split=val, stream_buffer=False, task=detect, time=None,
tracker=botsort.yaml, translate=0.1, val=True, verbose=True,
vid_stride=1, visualize=False, warmup_bias_lr=0.1, warmup_epochs=3.0,
warmup_momentum=0.8, weight_decay=0.0005, workers=8, workspace=None



Downloading https://ultralytics.com/assets/Arial.ttf to
'/root/.config/Ultralytics/Arial.ttf'...
```

```
# Load the best saved PyTorch model for prediction
# Define the model architecture (must match the saved model)
class PlantClassifier(nn.Module):
    def __init__(self, num_classes):
        super(PlantClassifier, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)

        # Calculate the size of the flattened layer dynamically
        # We'll use a dummy tensor to calculate the size after conv
and pool layers
        self._to_linear = None
        self.convs = nn.Sequential(
            nn.Conv2d(3, 16, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self._calculate_flatten_size()

        self.fc1 = nn.Linear(self._to_linear, 512)
        self.fc2 = nn.Linear(512, num_classes)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.3)

    def _calculate_flatten_size(self):
        # Pass a dummy tensor through the convolutional and pooling
layers
        x = torch.randn(1, 3, 224, 224) # Assuming input image size is
224x224
        x = self.convs(x)
        self._to_linear = x[0].shape[0] * x[0].shape[1] *
x[0].shape[2]


    def forward(self, x):
        x = self.convs(x)
        x = x.view(-1, self._to_linear) # Use the calculated flattened
size
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.fc2(x)
        return x

# Assume num_classes is 38 based on previous steps or load dynamically
if possible
# For now, assuming 38 based on the dataset used in training
num_classes = 38
```

```
plant_classifier_model = PlantClassifier(num_classes=num_classes)
try:

plant_classifier_model.load_state_dict(torch.load("best_model.pth",
map_location=device))
    plant_classifier_model.to(device)
    plant_classifier_model.eval()
    print("Plant classifier model loaded successfully from
best_model.pth for prediction")
except FileNotFoundError:
    print("Error: best_model.pth not found. Cannot load plant
classifier model.")
    plant_classifier_model = None # Set model to None if loading fails
except Exception as e:
    print(f"An error occurred while loading the plant classifier
model: {e}")
    plant_classifier_model = None


# Load the trained YOLOv8 model
yolo_model_path = os.path.join(yolo_extract_path,
'/content/yolov8n.pt') # Assuming the best model is saved here by
default
yolo_detection_model = None
if yolo_data_extracted and os.path.exists(yolo_model_path):
    try:
        yolo_detection_model = YOLO(yolo_model_path)
        print("YOLOv8 detection model loaded successfully.")
    except Exception as e:
        print(f"Error loading YOLOv8 detection model from
{yolo_model_path}: {e}")
else:
    print(f"Error: YOLOv8 model not found at {yolo_model_path} or YOLO
data not extracted. Cannot load YOLOv8 model.")
```

```
Plant classifier model loaded successfully from best_model.pth for
prediction
YOLOv8 detection model loaded successfully.
```

```
image_folder = "/content/drive/MyDrive/nit patna images"
import cv2
import numpy as np
import os
import matplotlib.pyplot as plt

rows, cols = 10, 6
img_h, img_w = 224, 224

images = []
loaded_images_count = 0
```

```
for i in range(1, rows * cols + 1):
    img_path = os.path.join(image_folder, f"image_{i}.jpg")
    img = cv2.imread(img_path)
    if img is not None:
        img = cv2.resize(img, (img_w, img_h))
        images.append(img)
        loaded_images_count += 1
    else:
        print(f"Warning: Could not load image {img_path}. Using
placeholder.")
        images.append(np.zeros((img_h, img_w, 3), dtype=np.uint8))

grid = []
if images:
    for i in range(rows):
        row_imgs = images[i * cols:(i + 1) * cols]
        if len(row_imgs) == cols:
            valid_row = True
            for img in row_imgs:
                if img.shape != (img_h, img_w, 3):
                    print(f"Error: Image in row {i+1} has incorrect
dimensions: {img.shape}")
                    valid_row = False
                    break
            if valid_row:
                try:
                    row = np.hstack(row_imgs)
                    grid.append(row)
                    print(f"Row {i+1} shape: {row.shape}") # Debug
print
                except ValueError as e:
                    print(f"Error stacking row {i+1}: {e}")
            else:
                print(f"Skipping row {i+1} due to invalid image
dimensions.")
        else:
            print(f"Warning: Incomplete row {i+1}. Expected {cols}
images, found {len(row_imgs)}. Skipping row.")
else:
    print("No images were loaded to create the grid.")

# You can now work with the 'grid' list of image rows.
# For example, to display the first row:
# if grid:
#     plt.imshow(cv2.cvtColor(grid[0], cv2.COLOR_BGR2RGB))
#     plt.axis('off')
#     plt.show()
```

```
Row 1 shape: (224, 1344, 3)
Row 2 shape: (224, 1344, 3)
```

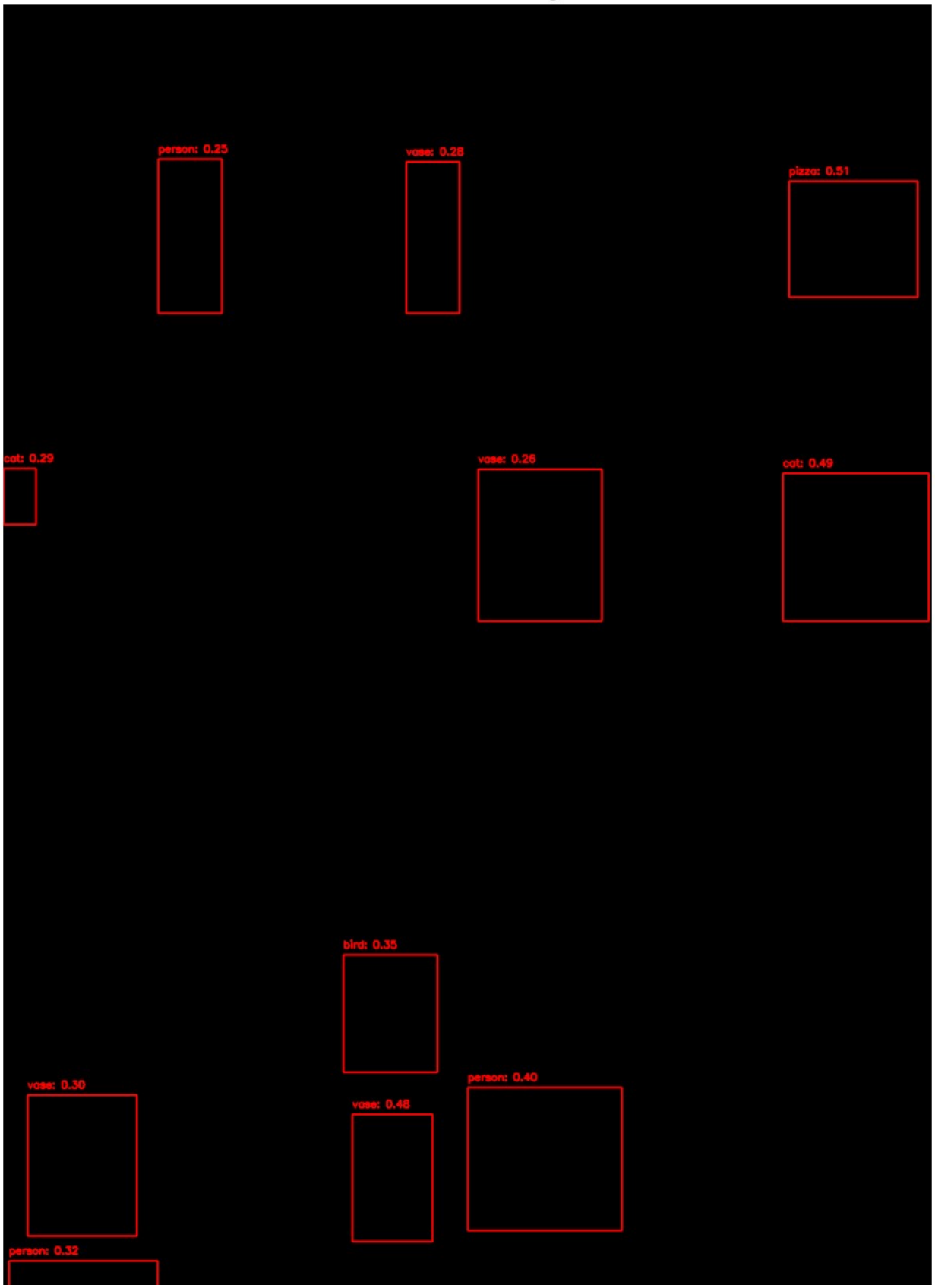
```
Row 3 shape: (224, 1344, 3)
Row 4 shape: (224, 1344, 3)
Row 5 shape: (224, 1344, 3)
Row 6 shape: (224, 1344, 3)
Row 7 shape: (224, 1344, 3)
Row 8 shape: (224, 1344, 3)
Row 9 shape: (224, 1344, 3)
Row 10 shape: (224, 1344, 3)
```

```
# Process mosaic with YOLOv8 for detection
if yolo_detection_model and grid:
    print("Processing image grid with YOLOv8 model...")

    # Create a blank canvas to draw detections on
    # Assuming all rows in the grid have the same width
    grid_height = sum([row.shape[0] for row in grid])
    grid_width = grid[0].shape[1] if grid else 0
    detection_grid = np.zeros((grid_height, grid_width, 3),
dtype=np.uint8)

    current_h = 0
    for row_idx, row_img in enumerate(grid):
        current_w = 0
        for col_idx in range(cols):
            # Extract the individual image from the row
            img = row_img[:, current_w : current_w + img_w]

            # Perform detection on the individual image
            results = yolo_detection_model(img)

            # Draw results on the individual image part of the
detection_grid
            # The coordinates from results are relative to the
individual image
            for r in results:
                boxes = r.boxes
                for box in boxes:
                    x1, y1, x2, y2 = box.xyxy[0].int().tolist()
                    conf = box.conf[0].item()
                    cls = box.cls[0].item()
                    class_name = yolo_detection_model.names[int(cls)]

                    # Adjust coordinates to the grid position
                    grid_x1 = x1 + current_w
                    grid_y1 = y1 + current_h
                    grid_x2 = x2 + current_w
                    grid_y2 = y2 + current_h

                    # Draw bounding box and label on the detection
grid
```

```
                    color = (0, 255, 0) if class_name == 'Healthy'
else (0, 0, 255) # Green for Healthy, Red for Unhealthy
                    cv2.rectangle(detection_grid, (grid_x1, grid_y1),
(grid_x2, grid_y2), color, 2)
                    label = f'{class_name}: {conf:.2f}'
                    cv2.putText(detection_grid, label, (grid_x1,
grid_y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

            current_w += img_w
        current_h += img_h

    # Display the grid with detections
    plt.figure(figsize=(20, 20))
    plt.imshow(cv2.cvtColor(detection_grid, cv2.COLOR_BGR2RGB))
    plt.title('YOLOv8 Detections on Image Grid')
    plt.axis('off')
    plt.show()

elif not yolo_detection_model:
    print("YOLOv8 detection model not loaded. Skipping detection on
image grid.")
else:
    print("Image grid is empty. Skipping detection on image grid.")
```

```
Processing image grid with YOLOv8 model...

0: 640x640 (no detections), 11.5ms
Speed: 4.1ms preprocess, 11.5ms inference, 2.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 13.4ms
Speed: 3.3ms preprocess, 13.4ms inference, 1.2ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 12.0ms
Speed: 3.2ms preprocess, 12.0ms inference, 1.1ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 13.4ms
Speed: 4.8ms preprocess, 13.4ms inference, 0.8ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 9.9ms
Speed: 3.7ms preprocess, 9.9ms inference, 0.9ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 11.7ms
Speed: 3.7ms preprocess, 11.7ms inference, 1.0ms postprocess per image
at shape (1, 3, 640, 640)
```

YOLOv8 Detections on Image Grid



```
# --- Drone Connection ---
# Attempt to connect to the drone, but proceed even if it fails
try:
    # Assuming the drone is simulated or connected via UDP
    drone = connect('udp:127.0.0.1:14550', wait_ready=False,
timeout=10) # Reduced timeout and wait_ready=False
    print(" Drone connection attempted.")
    # Check if drone object is valid after attempted connection
    if drone and drone.is_connected():
         print(" Drone connected successfully!")
    else:
         print(" Drone connection failed after attempt.")
         drone = None # Ensure drone is None if connection wasn't
successful
except APIException as e:
    print(f" Drone connection failed: {e}")
    drone = None
except Exception as e:
    print(f" An unexpected error occurred during drone connection:
{e}")
    drone = None

# --- Basic Localization Setup ---
# Define these values regardless of drone connection status
base_lat, base_lon = 27.2046, 77.4977
scale_x, scale_y = 1e-6, 1e-6 # These scaling factors might need
calibration

if drone:
    print(f"Using drone's location as base (if available), otherwise
using default base: Lat={base_lat}, Lon={base_lon}")
    # In a real scenario with a connected drone, you might update
base_lat/lon
    # with drone.location.global_frame.lat and
drone.location.global_frame.lon
else:
    print(f"Drone not connected, using default base location:
Lat={base_lat}, Lon={base_lon}")
```

```
WARNING:dronekit:Link timeout, no heartbeat in last 5 seconds
ERROR:dronekit.mavlink:Exception in MAVLink input loop
Traceback (most recent call last):
  File "/usr/local/lib/python3.11/dist-packages/dronekit/mavlink.py",
line 211, in mavlink_thread_in
    fn(self)
  File "/usr/local/lib/python3.11/dist-packages/dronekit/__init__.py",
line 1370, in listener
    raise APIException('No heartbeat in %s seconds, aborting.' %
dronekit.APIException: No heartbeat in 30 seconds, aborting.
```

```
 Drone connection failed: Timeout in initializing connection.
Drone not connected, using default base location: Lat=27.2046,
Lon=77.4977

# Map detections to grid coordinates and geographic locations
detected_plants_locations = [] # To store grid coordinates and
potentially geographic locations

if yolo_detection_model and grid:
    print("Mapping detections to grid coordinates and geographic
locations...")

    current_h = 0
    for row_idx, row_img in enumerate(grid):
        current_w = 0
        for col_idx in range(cols):
            # Extract the individual image from the row
            img = row_img[:, current_w : current_w + img_w]

            # Get detection results for the individual image
            results = yolo_detection_model(img)

            for r in results:
                boxes = r.boxes
                for box in boxes:
                    x1, y1, x2, y2 = box.xyxy[0].int().tolist()
                    cls = box.cls[0].item()
                    class_name = yolo_detection_model.names[int(cls)]

                    # Calculate center of the bounding box within the
individual image
                    center_x_img = (x1 + x2) // 2
                    center_y_img = (y1 + y2) // 2

                    # Calculate center of the bounding box within the
full grid
                    center_x_grid = center_x_img + current_w
                    center_y_grid = center_y_img + current_h

                    location_info = {
                        'class': class_name,
                        'grid_coords': (center_x_grid, center_y_grid),
                        'geographic_coords': None # Initialize
geographic coordinates
                    }

                    # Calculate approximate geographic coordinates
using the predefined base and scale
                    # This is a simple linear mapping and assumes the
grid is oriented correctly
```

```
                    approx_lat = base_lat + center_y_grid * scale_y
                    approx_lon = base_lon + center_x_grid * scale_x
                    location_info['geographic_coords'] = (approx_lat,
approx_lon)
                    print(f"Detected {class_name} at grid
({center_x_grid}, {center_y_grid}), mapped to approx geo
({approx_lat:.6f}, {approx_lon:.6f})")


                    detected_plants_locations.append(location_info)


            current_w += img_w
        current_h += img_h

    print(f"Mapping complete. Found {len(detected_plants_locations)}
plant detections.")
    # You can inspect detected_plants_locations to see the results
    # print(detected_plants_locations)

elif not yolo_detection_model:
    print("YOLOv8 detection model not loaded. Cannot map detections.")
else:
    print("Image grid is empty. Cannot map detections.")
```

```
Mapping detections to grid coordinates and geographic locations...

0: 640x640 (no detections), 13.4ms
Speed: 4.7ms preprocess, 13.4ms inference, 0.8ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 7.9ms
Speed: 2.3ms preprocess, 7.9ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 9.2ms
Speed: 2.7ms preprocess, 9.2ms inference, 0.7ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 7.9ms
Speed: 3.0ms preprocess, 7.9ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 8.1ms
Speed: 3.5ms preprocess, 8.1ms inference, 1.0ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 7.9ms
Speed: 2.9ms preprocess, 7.9ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)
```

```
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 6.6ms
Speed: 3.2ms preprocess, 6.6ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 6.8ms
Speed: 2.9ms preprocess, 6.8ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 6.3ms
Speed: 3.0ms preprocess, 6.3ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 6.3ms
Speed: 2.7ms preprocess, 6.3ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 6.2ms
Speed: 3.0ms preprocess, 6.2ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 1 cake, 6.3ms
Speed: 2.7ms preprocess, 6.3ms inference, 1.4ms postprocess per image
at shape (1, 3, 640, 640)
Detected cake at grid (569, 2130), mapped to approx geo (27.206730,
77.498269)

0: 640x640 (no detections), 6.4ms
Speed: 3.2ms preprocess, 6.4ms inference, 0.7ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 6.3ms
Speed: 3.1ms preprocess, 6.3ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)

0: 640x640 (no detections), 6.7ms
Speed: 2.8ms preprocess, 6.7ms inference, 0.6ms postprocess per image
at shape (1, 3, 640, 640)
Mapping complete. Found 12 plant detections.

# Visualize infected areas on the grid/map

if grid and detected_plants_locations:
    print("Visualizing infected areas...")

    # Start with the original image grid for visualization
    visualization_grid = np.copy(grid[0])
    for row_img in grid[1:]:
        visualization_grid = np.vstack((visualization_grid, row_img))
```

```
    # Draw indicators for detected plants, highlighting unhealthy ones
    for plant_info in detected_plants_locations:
        grid_x, grid_y = plant_info['grid_coords']
        class_name = plant_info['class']

        color = (0, 255, 0) # Green for Healthy
        # Use a different marker type or draw manually if
MARKER_CIRCLE is not available
        # marker_type = cv2.MARKER_CIRCLE # This might not be
available in older OpenCV versions
        marker_size = 10
        thickness = 2

        if class_name != 'Healthy':
            color = (0, 0, 255) # Red for Unhealthy
            # marker_type = cv2.MARKER_X # This might not be available
in older OpenCV versions
            marker_size = 15
            thickness = 3

        # Draw a marker at the center of the detected plant location
on the grid
        # Using cv2.circle for a circle marker or cv2.drawMarker with
available types
        # If MARKER_CIRCLE is not available, drawing a circle manually
is a good alternative
        cv2.circle(visualization_grid, (grid_x, grid_y), marker_size,
color, thickness)


        # Optionally, add text label with class name next to the
marker
        # cv2.putText(visualization_grid, class_name, (grid_x + 15,
grid_y + 5),
        #             cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 1)


    plt.figure(figsize=(20, 20))
    plt.imshow(cv2.cvtColor(visualization_grid, cv2.COLOR_BGR2RGB))
    plt.title('Visualizing Detected Plants (Markers: Green=Healthy,
Red=Unhealthy)')
    plt.axis('off')
    plt.show()

    # --- Optional: Visualize on a simple map if geographic data is
available ---
    # This is a very basic visualization. For a real application,
consider using
    # a proper mapping library like Folium or plotting on a more
```

```
accurate representation.

    unhealthy_locations = [loc for loc in detected_plants_locations if
loc['class'] != 'Healthy' and loc['geographic_coords'] is not None]

    if unhealthy_locations:
        print(f"\nVisualizing {len(unhealthy_locations)} unhealthy
plant locations on a simple map...")

        # Extract latitudes and longitudes
        lats = [loc['geographic_coords'][0] for loc in
unhealthy_locations]
        lons = [loc['geographic_coords'][1] for loc in
unhealthy_locations]

        plt.figure(figsize=(10, 8))
        plt.scatter(lons, lats, color='red', marker='X', s=100,
label='Unhealthy Plants')
        plt.xlabel("Approximate Longitude")
        plt.ylabel("Approximate Latitude")
        plt.title("Approximate Locations of Unhealthy Plants")
        plt.legend()
        plt.grid(True)
        plt.show()
    elif drone and detected_plants_locations:
         print("\nNo unhealthy plants detected with geographic
coordinates to visualize on a map.")
    elif not drone:
         print("\nDrone not connected, skipping geographic map
visualization.")

elif not grid:
    print("Image grid is empty. Skipping visualization.")
elif not detected_plants_locations:
    print("No plant detections found. Skipping visualization.")

Visualizing infected areas...
```

```
Drone not connected, skipping geographic map visualization.

# Identify disease type in infected areas

if plant_classifier_model and grid and detected_plants_locations:
    print("Identifying disease types in infected areas...")

    infected_plant_diseases = []

    # Prepare the full visualization grid to extract patches
    full_grid_img = np.copy(grid[0])
    for row_img in grid[1:]:
        full_grid_img = np.vstack((full_grid_img, row_img))

    for plant_info in detected_plants_locations:
        class_name = plant_info['class']

        # Only classify if the plant is detected as 'Unhealthy' by
YOLO
        # Assuming 'Healthy' is the only class indicating no disease
in YOLO output
        if class_name != 'Healthy':
            grid_x, grid_y = plant_info['grid_coords']

            # Assuming the detected plant is roughly centered around
the grid_x, grid_y
            # We need to extract a patch around these coordinates from
the full grid image.
            # The size of the patch should be appropriate for the CNN
model input (224x224).
            # We need to be careful with boundary conditions.

            # Calculate the top-left corner of the patch
            patch_size = 224 # Assuming CNN input size is 224x224
            half_patch = patch_size // 2

            # Ensure the patch coordinates are within the bounds of
the full grid image
            start_y = max(0, grid_y - half_patch)
            end_y = min(full_grid_img.shape[0], grid_y + half_patch)
            start_x = max(0, grid_x - half_patch)
            end_x = min(full_grid_img.shape[1], grid_x + half_patch)

            # Extract the patch
            # Adjust patch size if near boundaries
            actual_patch_h = end_y - start_y
            actual_patch_w = end_x - start_x

            if actual_patch_h > 0 and actual_patch_w > 0:
                plant_patch = full_grid_img[start_y:end_y,
```

```
start_x:end_x]

                # Resize the patch to the expected CNN input size if
it's not already
                if plant_patch.shape[0] != patch_size or
plant_patch.shape[1] != patch_size:
                    plant_patch_resized = cv2.resize(plant_patch,
(patch_size, patch_size))
                else:
                    plant_patch_resized = plant_patch


                # Convert the patch to PIL Image and apply the
prediction transform
                try:
                    # Ensure the image is in the correct format (RGB)
                    plant_patch_pil =
Image.fromarray(cv2.cvtColor(plant_patch_resized, cv2.COLOR_BGR2RGB))
                    input_tensor =
predict_transform(plant_patch_pil).unsqueeze(0).to(device)

                    # Get disease prediction from the classification
model
                    with torch.no_grad():
                        outputs = plant_classifier_model(input_tensor)
                        _, predicted_idx = torch.max(outputs, 1)
                        predicted_disease =
class_names[predicted_idx.item()]

                    infected_plant_diseases.append({
                        'grid_coords': (grid_x, grid_y),
                        'predicted_disease': predicted_disease,
                        'geographic_coords':
plant_info['geographic_coords'] # Include geographic coords
                    })
                    print(f"Identified disease at grid ({grid_x},
{grid_y}): {predicted_disease}")

                except Exception as e:
                    print(f"Error processing patch at grid ({grid_x},
{grid_y}) for disease classification: {e}")
            else:
                 print(f"Skipping disease classification for detection
at grid ({grid_x}, {grid_y}) due to invalid patch size
({actual_patch_w}x{actual_patch_h}).")


    print(f"Disease identification complete. Identified diseases for
{len(infected_plant_diseases)} infected plant detections.")
```

```
    # You can now use the 'infected_plant_diseases' list for further
visualization or analysis.
    # For example, drawing the disease name on the visualization grid.

elif not plant_classifier_model:
    print("Plant classifier model not loaded. Cannot identify disease
types.")
elif not grid:
    print("Image grid is empty. Cannot identify disease types.")
elif not detected_plants_locations:
    print("No plant detections found. Cannot identify disease types.")
```

```
Identifying disease types in infected areas...
Identified disease at grid (270, 335): Corn_(maize)___Common_rust_
Identified disease at grid (621, 337):
Corn_(maize)___Northern_Leaf_Blight
Identified disease at grid (1230, 340): Peach___Bacterial_spot
Identified disease at grid (23, 712): Strawberry___Leaf_scorch
Identified disease at grid (776, 783): Pepper,_bell___Bacterial_spot
Identified disease at grid (1233, 786): Pepper,_bell___healthy
Identified disease at grid (560, 1461): Blueberry___healthy
Identified disease at grid (114, 1681):
Cherry_(including_sour)___healthy
Identified disease at grid (563, 1699): Tomato___Septoria_leaf_spot
Identified disease at grid (783, 1671): Apple___Apple_scab
Identified disease at grid (115, 1917): Peach___Bacterial_spot
Identified disease at grid (569, 2130): Strawberry___Leaf_scorch
Disease identification complete. Identified diseases for 12 infected
plant detections.
```

```
# Display Visualization of Detected Plants on Grid
if 'visualization_grid' in locals() and visualization_grid is not
None:
    print("Displaying Visualization of Detected Plants on Image
Grid:")
    # Start with the original image grid for visualization
    visualization_grid = np.copy(grid[0])
    for row_img in grid[1:]:
        visualization_grid = np.vstack((visualization_grid, row_img))


    # Draw indicators for detected plants, highlighting unhealthy ones
and adding disease name
    if 'infected_plant_diseases' in locals():
        for plant_info in detected_plants_locations:
            grid_x, grid_y = plant_info['grid_coords']
            class_name = plant_info['class']

            color = (0, 255, 0) # Green for Healthy
            marker_size = 10
```

```
            thickness = 2
            # marker_type = cv2.MARKER_CIRCLE # Default marker


            if class_name != 'Healthy':
                color = (0, 0, 255) # Red for Unhealthy
                marker_size = 15
                thickness = 3
                # marker_type = cv2.MARKER_X # Use 'X' marker for
unhealthy plants

                # Find the predicted disease for this unhealthy plant
                predicted_disease = "Unknown Disease"
                for infected_info in infected_plant_diseases:
                    if infected_info['grid_coords'] == (grid_x,
grid_y):
                        predicted_disease =
infected_info['predicted_disease']
                        break

                # Add text label with disease name next to the marker
                cv2.putText(visualization_grid, predicted_disease,
(grid_x + 15, grid_y + 5),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 1)

            # Draw the marker
            # Check if cv2.drawMarker is available (requires OpenCV
3.2+)
            # if hasattr(cv2, 'drawMarker'):
            #     cv2.drawMarker(visualization_grid, (grid_x,
grid_y), color,
            #                    markerType=marker_type,
            #                    markerSize=marker_size,
            #                    thickness=thickness)
            # else:
                # Fallback to drawing a circle if drawMarker is not
available
            cv2.circle(visualization_grid, (grid_x, grid_y),
marker_size, color, thickness)


    plt.figure(figsize=(20, 20))
    plt.imshow(cv2.cvtColor(visualization_grid, cv2.COLOR_BGR2RGB))
    plt.title('Visualizing Detected Plants and Diseases
(Green=Healthy, Red=Unhealthy with Disease Label)')
    plt.axis('off')
    plt.show()

elif grid and detected_plants_locations:
     print("Visualization grid not explicitly saved, but detected
```

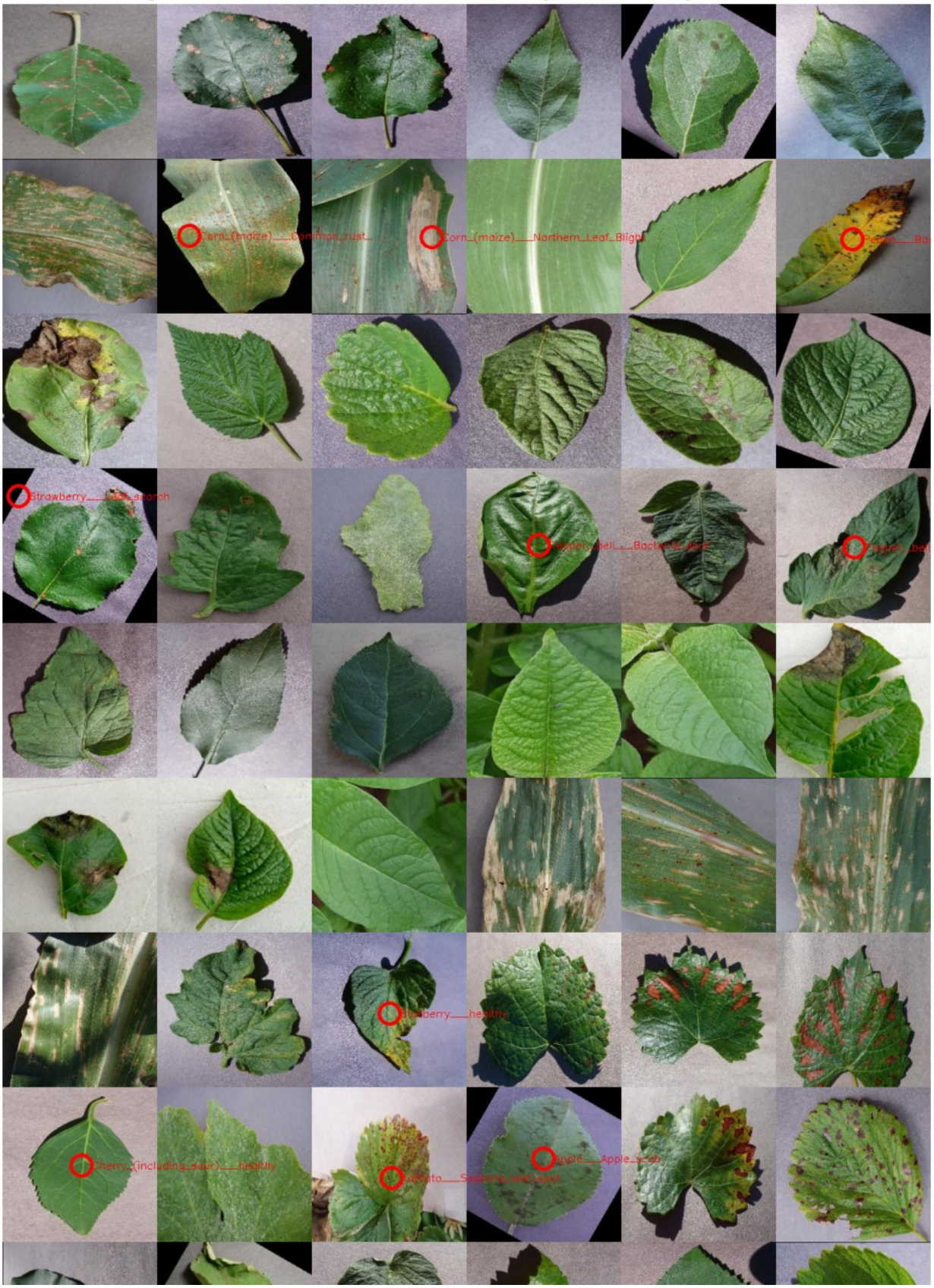
```
plants and grid are available. Re-generating basic visualization...")
    # Re-generate a basic visualization if the variable wasn't saved
    temp_vis_grid = np.copy(grid[0])
    for row_img in grid[1:]:
        temp_vis_grid = np.vstack((temp_vis_grid, row_img))

    if 'infected_plant_diseases' in locals():
        for plant_info in detected_plants_locations:
            grid_x, grid_y = plant_info['grid_coords']
            class_name = plant_info['class']

            color = (0, 255, 0) # Green for Healthy
            # marker_type = cv2.MARKER_CIRCLE
            marker_size = 10
            thickness = 2

            if class_name != 'Healthy':
                color = (0, 0, 255) # Red for Unhealthy
                # marker_type = cv2.MARKER_X # Use 'X' marker for
unhealthy plants
                marker_size = 15
                thickness = 3

                # Find the predicted disease for this unhealthy plant
                predicted_disease = "Unknown Disease"
                for infected_info in infected_plant_diseases:
                    if infected_info['grid_coords'] == (grid_x,
grid_y):
                        predicted_disease =
infected_info['predicted_disease']
                        break

                # Add text label with disease name next to the marker
                cv2.putText(temp_vis_grid, predicted_disease, (grid_x
+ 15, grid_y + 5),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 1)

            # Draw the marker
            # if hasattr(cv2, 'drawMarker'):
            #     cv2.drawMarker(temp_vis_grid, (grid_x, grid_y),
color,
            #                    markerType=marker_type,
            #                    markerSize=marker_size,
            #                    thickness=thickness)
            # else:
            cv2.circle(temp_vis_grid, (grid_x, grid_y), marker_size,
color, thickness)


    plt.figure(figsize=(20, 20))
```

```
     plt.imshow(cv2.cvtColor(temp_vis_grid, cv2.COLOR_BGR2RGB))
     plt.title('Visualizing Detected Plants and Diseases
(Green=Healthy, Red=Unhealthy with Disease Label)')
     plt.axis('off')
     plt.show()

else:
    print("No visualization grid available to display.")

# Display Simple Map Visualization (if geographic data is available)
# Use infected_plant_diseases which now includes geographic
coordinates
if 'infected_plant_diseases' in locals() and infected_plant_diseases:
    unhealthy_infected_locations = [loc for loc in
infected_plant_diseases if loc['geographic_coords'] is not None]

    if unhealthy_infected_locations:
        print(f"\nDisplaying Approximate Locations of Unhealthy Plants
with Diseases on a simple map...")

        # Extract latitudes, longitudes, and disease names
        lats = [loc['geographic_coords'][0] for loc in
unhealthy_infected_locations]
        lons = [loc['geographic_coords'][1] for loc in
unhealthy_infected_locations]
        disease_names = [loc['predicted_disease'] for loc in
unhealthy_infected_locations]

        plt.figure(figsize=(10, 8))
        scatter = plt.scatter(lons, lats, color='red', marker='X',
s=100, label='Unhealthy Plants')

        # Add text labels for disease names
        for i, txt in enumerate(disease_names):
            plt.annotate(txt, (lons[i], lats[i]), textcoords="offset
points", xytext=(10,10), ha='center')


        plt.xlabel("Approximate Longitude")
        plt.ylabel("Approximate Latitude")
        plt.title("Approximate Locations and Diseases of Unhealthy
Plants")
        plt.legend()
        plt.grid(True)
        plt.show()
    else:
         print("\nNo unhealthy plants with identified diseases and
geographic coordinates to visualize on a map.")

elif detected_plants_locations and any(loc['class'] != 'Healthy' for
```

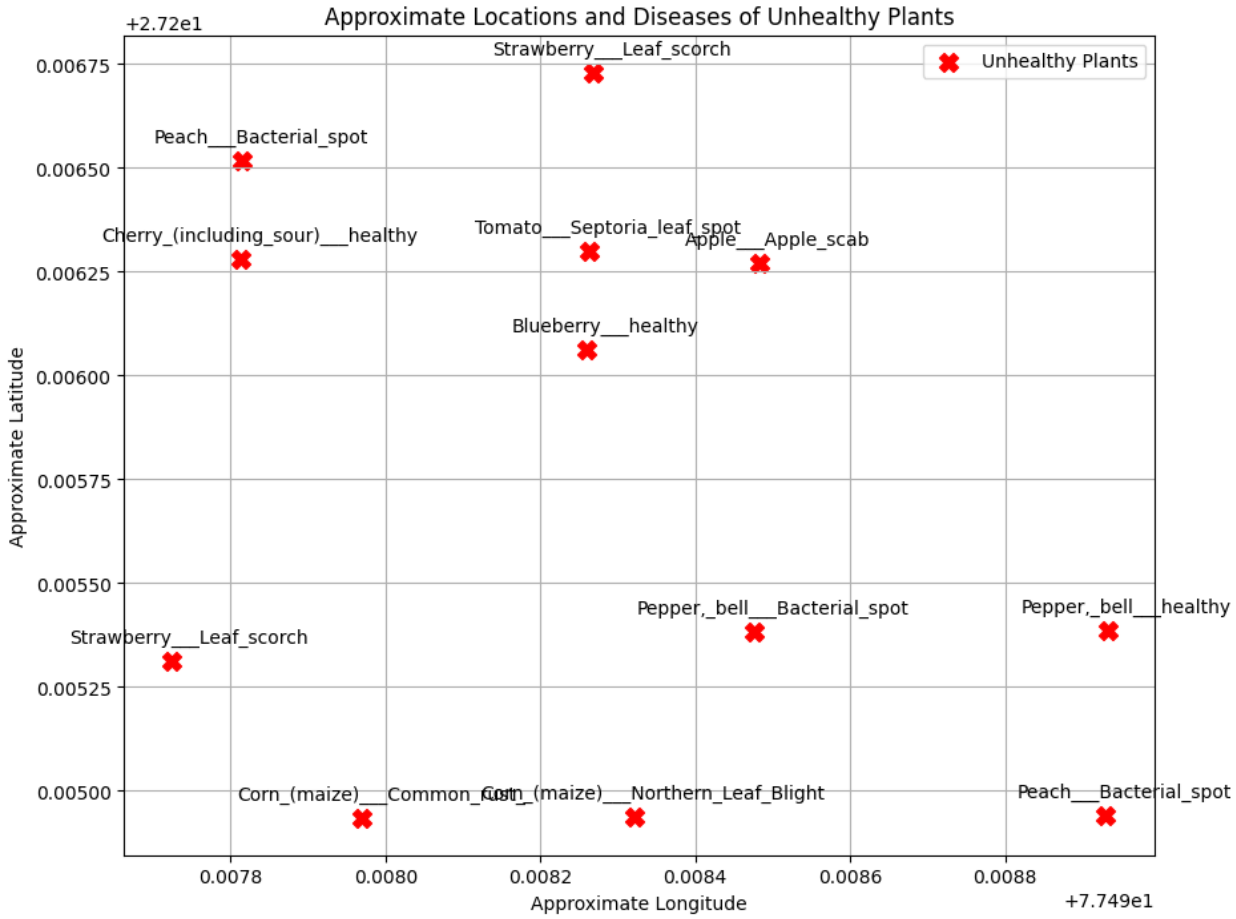
```
loc in detected_plants_locations):
     print("\nInfected plants detected, but disease identification
results are not available for map visualization (ensure the previous
step ran successfully and produced geographic coordinates).")
else:
    print("\nNo infected plants were detected or disease
identification was not performed for map visualization.")


# Display Identified Diseases (from previous step's output)
if 'infected_plant_diseases' in locals() and infected_plant_diseases:
    print("\nIdentified Disease Types in Infected Areas:")
    for disease_info in infected_plant_diseases:
        print(f" - At grid coordinates {disease_info['grid_coords']}:
{disease_info['predicted_disease']}")
else:
    print("\nNo infected plants were detected or disease
identification was not performed.")

# Resource Links (Placeholder)
print("\n--- Resource Links ---")
print("- PlantVillage Dataset: [Link to dataset]")
print("- YOLOv8 Documentation: [Link to documentation]")
print("- PyTorch Documentation: [Link to PyTorch documentation]")
print("- DroneKit Documentation: [Link to DroneKit documentation]")
print("\nPlease replace the bracketed links above with the actual URLs
of the resources you used.")
```

```
Displaying Visualization of Detected Plants on Image Grid:
```

Visualizing Detected Plants and Diseases (Green=Healthy, Red=Unhealthy with Disease Label)



```
Displaying Approximate Locations of Unhealthy Plants with Diseases on
a simple map...
```




```
Identified Disease Types in Infected Areas:
 - At grid coordinates (270, 335): Corn_(maize)___Common_rust_
 - At grid coordinates (621, 337): Corn_(maize)___Northern_Leaf_Blight
 - At grid coordinates (1230, 340): Peach___Bacterial_spot
 - At grid coordinates (23, 712): Strawberry___Leaf_scorch
 - At grid coordinates (776, 783): Pepper,_bell___Bacterial_spot
 - At grid coordinates (1233, 786): Pepper,_bell___healthy
 - At grid coordinates (560, 1461): Blueberry___healthy
 - At grid coordinates (114, 1681): Cherry_(including_sour)___healthy
 - At grid coordinates (563, 1699): Tomato___Septoria_leaf_spot
 - At grid coordinates (783, 1671): Apple___Apple_scab
 - At grid coordinates (115, 1917): Peach___Bacterial_spot
 - At grid coordinates (569, 2130): Strawberry___Leaf_scorch

--- Resource Links ---
- PlantVillage Dataset: [Link to dataset]
```

```
- YOLOv8 Documentation: [Link to documentation]
- PyTorch Documentation: [Link to PyTorch documentation]
- DroneKit Documentation: [Link to DroneKit documentation]

Please replace the bracketed links above with the actual URLs of the resources you used.
```