Data-Driven Crop Diagnostic Platform for Farmers

Project Overview

Our project aims to develop a data-driven AI platform that leverages image classification technology to diagnose crop diseases and provide comprehensive insights and actionable recommendations to rural farmers. By integrating image data with environmental factors, we strive to enhance agricultural productivity and promote sustainable farming practices.

Objectives

- **Identify Crop Diseases**: Use image classification to accurately identify various crop diseases.
- **Predictive Model Development**: Develop a model that combines image data with environmental factors to provide comprehensive insights into crop health.
- **Actionable Recommendations**: Provide farmers with practical advice on managing and preventing crop diseases based on the model's predictions.
- **Time Series Analysis**: Investigate the use of time series analysis to predict the spread of crop diseases based on environmental factors.

Our Solution

AI-Driven Crop Diagnostic Platform: A user-friendly mobile application where farmers can upload images of their crops. Our AI algorithms will analyze the images to diagnose issues such as pests, diseases, and nutrient deficiencies. The platform will also incorporate environmental data to provide a holistic view of crop health.

Key Features:

- **Instant Diagnostics**: Real-time analysis of crop images with high accuracy.
- **Predictive Insights**: Combining image data with environmental factors for comprehensive health insights.
- **Actionable Recommendations**: Tailored advice on how to address identified issues and prevent future occurrences.
- **Support Network**: Access to community forums, real-time chat support, and local field agents for additional assistance.

Data Sources Details

PlantVillage Dataset

- **Source**: Kaggle, PlantVillage
- **Columns**:

`image_id`: Unique identifier for each image

`image_path`: Path to the image file

`crop_type`: Type of crop (e.g., apple, grape, tomato)

`disease_type`: Type of disease (e.g., apple scab, grape black rot)

`health_status`: Health status of the plant (healthy or diseased)

`image`: The image data itself

Mendeley Data Repository

- **Source**: Mendeley Data Repository
- **Columns**: **`image_id`**: Unique identifier for each image

`image_path`: Path to the image file

`crop_type`: Type of crop

`disease_type`: Type of disease or pest

`nutrient_deficiency`: Type of nutrient deficiency (if applicable) **`location`**: Geographic location where the image was taken

`image`: The image data itself

NASA Earth Observing System Data and Information System (EOSDIS)

- **Source**: NASA EOSDIS
- **Columns**: **`date`**: Date of the observation

`latitude`: Latitude of the observation location

`longitude`: Longitude of the observation location

`temperature`: Surface temperature (in Celsius)

`precipitation`: Precipitation levels (in mm)

`humidity`: Relative humidity (in %)

`solar_radiation`: Solar radiation levels (in W/m^2)

`soil_moisture`: Soil moisture levels (in m^3/m^3)

Data Loading and Understanding

Load the images from both folders using a data generator and displays the names of all the classes and also images from the various classes.

```
# import libraries
import os
```

```
import glob
import re
import matplotlib.pyplot as plt
from PIL import Image
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import shutil
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.preprocessing import image
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.utils.class_weight import compute_class_weight
from tensorflow.keras.layers import Input
from tensorflow.keras.models import load_model
from tensorflow.keras.callbacks import EarlyStopping
```

Create a class **`DataLoadingAndUnderstanding`** to `load`, `manage` and `visualize image data` from directories for image classification tasks, specifically for datasets with and without augmentation.

```
# Class DataLoadingAndUnderstanding
class DataLoadingAndUnderstanding:
    def __init__(self, no_aug_dir, target_size=(224, 224),
batch_size=32):
        """
        Initialize with directories containing the dataset, and set up
data generators.

        :param no_aug_dir: Directory for dataset without augmentation.
        :param aug_dir: Directory for dataset with augmentation.
        :param target_size: Tuple specifying the target size of the
images.
        :param batch_size: Batch size for data generators.
        """
        self.no_aug_dir = no_aug_dir
        self.target_size = target_size
        self.batch_size = batch_size

        # Initialize data generators
        self.no_aug_datagen = ImageDataGenerator(rescale=1./255)

        # Load data
        self.no_aug_generator = self._load_data(self.no_aug_datagen,
self.no_aug_dir)
```

```
        # Get classes and convert to list
        self.classes =
list(self.no_aug_generator.class_indices.keys())

    def _load_data(self, datagen, directory):
        """
        Private method to load data from a directory using a given
ImageDataGenerator.

        :param datagen: Instance of ImageDataGenerator.
        :param directory: Directory from which to load images.
        :return: A generator that yields batches of images from the
specified directory.
        """
        return datagen.flow_from_directory(
            directory,
            target_size=self.target_size,
            batch_size=self.batch_size,
            class_mode='categorical'
        )

    def show_classes(self):
        """Display the list of classes (disease categories)."""
        print(f"Classes: {list(self.classes)}")

    def visualize_images_from_folder(self, folder_path,
num_images=10):
        """
        Display a few images from a specified folder.

        :param folder_path: Path to the folder containing images.
        :param num_images: Number of images to display.
        """
        # Ensure the folder exists
        if not os.path.exists(folder_path):
            print(f"Folder does not exist: {folder_path}")
            return

        # List all image files in the folder
        image_files = [f for f in os.listdir(folder_path) if
f.lower().endswith(('.jpg', '.jpeg', '.png'))]

        # Check if there are images available
        if len(image_files) == 0:
            print(f"No images found in {folder_path}")
            return

        # Limit to the specified number of images
        image_files = image_files[:num_images]
```

```
        plt.figure(figsize=(15, 15))

        for i, image_file in enumerate(image_files):
            image_path = os.path.join(folder_path, image_file)
            try:
                img = Image.open(image_path)
                plt.subplot(1, num_images, i + 1)
                plt.imshow(img)
                plt.axis('off')
            except Exception as e:
                print(f"Could not open image {image_path}: {e}")

        plt.tight_layout()
        plt.show()

    def display_sample_images(self, samples_per_class=1,
image_size=(224, 224)):
        """
        Display sample images from each class to get an understanding
of the data.

        :param samples_per_class: Number of images to display per
class.
        :param image_size: Size to resize the images for display
(width, height).
        """
        num_classes = len(self.classes)
        fig, axes = plt.subplots(nrows=num_classes,
ncols=samples_per_class, figsize=(5 * samples_per_class, 5 *
num_classes))

        # Loop through each class and display sample images
        for i, cls in enumerate(self.classes):
            image_paths = glob.glob(os.path.join(self.no_aug_dir, cls,
"*.JPG"))
            if len(image_paths) == 0:
                # If there are no images for this class, skip it and
continue
                print(f"No images found for class: {cls}")
                continue

            for j in range(min(samples_per_class, len(image_paths))):
# Display only available images
                image_path = image_paths[j]  # Get the j-th image for
the i-th class
                image = Image.open(image_path).resize(image_size)  #
Resize image
                axes[i, j].imshow(image)
                axes[i, j].set_title(cls, fontsize=12)  # Set the
title as the class name
```

```
                axes[i, j].axis('off')  # Hide axis

            # Hide unused subplots
            for k in range(len(image_paths), samples_per_class):
                axes[i, k].axis('off')

        plt.tight_layout()
        plt.show()

# Visualize image from specific folder
data_loader = DataLoadingAndUnderstanding(
    no_aug_dir='Plant_leave_diseases_dataset_without_augmentation'
)
```

```
Found 54300 images belonging to 38 classes.
```

```
data_loader.show_classes()
```

```
Classes: ['Apple___Apple_scab', 'Apple___Black_rot',
'Apple___Cedar_apple_rust', 'Apple___healthy', 'Blueberry___healthy',
'Cherry___Powdery_mildew', 'Cherry___healthy',
'Corn___Cercospora_leaf_spot Gray_leaf_spot', 'Corn___Common_rust',
'Corn___Northern_Leaf_Blight', 'Corn___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']
```

```
# Example to see if it works
data_loader.visualize_images_from_folder('Plant_leave_diseases_dataset
_without_augmentation/Apple___Apple_scab')
```

```
data_loader.display_sample_images(samples_per_class=2)
```

Apple___Apple_scab

Apple___Apple_scab

Apple___Black_rot

Apple___Black_rot

Apple___Cedar_apple_rust

Apple___Cedar_apple_rust

3. Data Cleaning

3.1 Remove Corrupt Images

Check for and remove any corrupt or unreadable images.

```
# Create function `check_corrupt_images`
def check_corrupt_images(directory):
    corrupt_images = []

    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.lower().endswith(('.jpg', '.jpeg', '.png')):
                file_path = os.path.join(root, file)
                try:
                    # Attempt to open the image and verify it
                    with Image.open(file_path) as img:
                        img.verify()  # Verify that it is not corrupt
                except (IOError, SyntaxError, OSError) as e:
                    print(f'Corrupt image detected and removed:
{file_path}')
                    os.remove(file_path)  # Remove the corrupt image
                    corrupt_images.append(file_path)

    if not corrupt_images:
        return "No corrupt images found."
    else:
        return f"Total corrupt images found and removed:
{len(corrupt_images)}"
```

This function identifies and removes corrupt image files from a specified directory.

```
# Specify the directories to the datasets
no_aug_dir='Plant_leave_diseases_dataset_without_augmentation'

# Call the function to check  for corrupt images in the dataset
without_augmentation
result = check_corrupt_images(no_aug_dir)
print(result)
```

```
No corrupt images found.
```

There are no corrupt images in our dataset

Rename the classes

```
# Function to clean class names
def clean_class_names(classes):
    cleaned_classes = []
```

```
    for name in classes:
        # Remove underscores and dashes, and split the name into words
        cleaned_name = re.sub(r'[_-]', ' ', name)
        words = cleaned_name.split()
        # Remove repeated words and rejoin them
        cleaned_name = ' '.join(sorted(set(words), key=words.index))
        cleaned_classes.append(cleaned_name)
    return cleaned_classes

# List of class names to clean
classes = ['Apple___Apple_scab', 'Apple___Black_rot',
'Apple___Cedar_apple_rust', 'Apple___healthy',
           'Blueberry___healthy', 'Cherry___Powdery_mildew',
'Cherry___healthy',
           'Corn___Cercospora_leaf_spot Gray_leaf_spot',
'Corn___Common_rust', 'Corn___Northern_Leaf_Blight',
           'Corn___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']

# Clean the class names
clean_classes = clean_class_names(classes)
print("Clean Classes:", clean_classes)
```

```
Clean Classes: ['Apple scab', 'Apple Black rot', 'Apple Cedar apple
rust', 'Apple healthy', 'Blueberry healthy', 'Cherry Powdery mildew',
'Cherry healthy', 'Corn Cercospora leaf spot Gray', 'Corn Common
rust', 'Corn Northern Leaf Blight', 'Corn healthy', 'Grape Black rot',
'Grape Esca (Black Measles)', 'Grape Leaf blight (Isariopsis 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 Yellow Leaf
Curl Virus', 'Tomato mosaic virus', 'Tomato healthy']

# Create a mapping of original to cleaned names
class_name_mapping = dict(zip(classes, clean_classes))

# Path to the directory containing the folders
base_dir = 'Plant_leave_diseases_dataset_without_augmentation'

# Rename folders
for original_name, cleaned_name in class_name_mapping.items():
    original_folder_path = os.path.join(base_dir, original_name)
    cleaned_folder_path = os.path.join(base_dir, cleaned_name)

    if os.path.exists(original_folder_path):
        os.rename(original_folder_path, cleaned_folder_path)
        print(f"Renamed '{original_name}' to '{cleaned_name}'")
    else:
        print(f"Folder '{original_name}' does not exist")

print("Renaming complete.")

Renamed 'Apple___Apple_scab' to 'Apple scab'
Renamed 'Apple___Black_rot' to 'Apple Black rot'
Renamed 'Apple___Cedar_apple_rust' to 'Apple Cedar apple rust'
Renamed 'Apple___healthy' to 'Apple healthy'
Renamed 'Blueberry___healthy' to 'Blueberry healthy'
Renamed 'Cherry___Powdery_mildew' to 'Cherry Powdery mildew'
Renamed 'Cherry___healthy' to 'Cherry healthy'
Renamed 'Corn___Cercospora_leaf_spot Gray_leaf_spot' to 'Corn
Cercospora leaf spot Gray'
Renamed 'Corn___Common_rust' to 'Corn Common rust'
Renamed 'Corn___Northern_Leaf_Blight' to 'Corn Northern Leaf Blight'
Renamed 'Corn___healthy' to 'Corn healthy'
Renamed 'Grape___Black_rot' to 'Grape Black rot'
Renamed 'Grape___Esca_(Black_Measles)' to 'Grape Esca (Black Measles)'
Renamed 'Grape___Leaf_blight_(Isariopsis_Leaf_Spot)' to 'Grape Leaf
blight (Isariopsis Spot)'
Renamed 'Grape___healthy' to 'Grape healthy'
Renamed 'Orange___Haunglongbing_(Citrus_greening)' to 'Orange
Haunglongbing (Citrus greening)'
Renamed 'Peach___Bacterial_spot' to 'Peach Bacterial spot'
Renamed 'Peach___healthy' to 'Peach healthy'
Renamed 'Pepper,_bell___Bacterial_spot' to 'Pepper, bell Bacterial
spot'
Renamed 'Pepper,_bell___healthy' to 'Pepper, bell healthy'
Renamed 'Potato___Early_blight' to 'Potato Early blight'
```

```
Renamed 'Potato___Late_blight' to 'Potato Late blight'
Renamed 'Potato___healthy' to 'Potato healthy'
Renamed 'Raspberry___healthy' to 'Raspberry healthy'
Renamed 'Soybean___healthy' to 'Soybean healthy'
Renamed 'Squash___Powdery_mildew' to 'Squash Powdery mildew'
Renamed 'Strawberry___Leaf_scorch' to 'Strawberry Leaf scorch'
Renamed 'Strawberry___healthy' to 'Strawberry healthy'
Renamed 'Tomato___Bacterial_spot' to 'Tomato Bacterial spot'
Renamed 'Tomato___Early_blight' to 'Tomato Early blight'
Renamed 'Tomato___Late_blight' to 'Tomato Late blight'
Renamed 'Tomato___Leaf_Mold' to 'Tomato Leaf Mold'
Renamed 'Tomato___Septoria_leaf_spot' to 'Tomato Septoria leaf spot'
Renamed 'Tomato___Spider_mites Two-spotted_spider_mite' to 'Tomato
Spider mites Two spotted spider mite'
Renamed 'Tomato___Target_Spot' to 'Tomato Target Spot'
Renamed 'Tomato___Tomato_Yellow_Leaf_Curl_Virus' to 'Tomato Yellow
Leaf Curl Virus'
Renamed 'Tomato___Tomato_mosaic_virus' to 'Tomato mosaic virus'
Renamed 'Tomato___healthy' to 'Tomato healthy'
Renaming complete.
```

4. Exploratory Data Analysis (EDA)

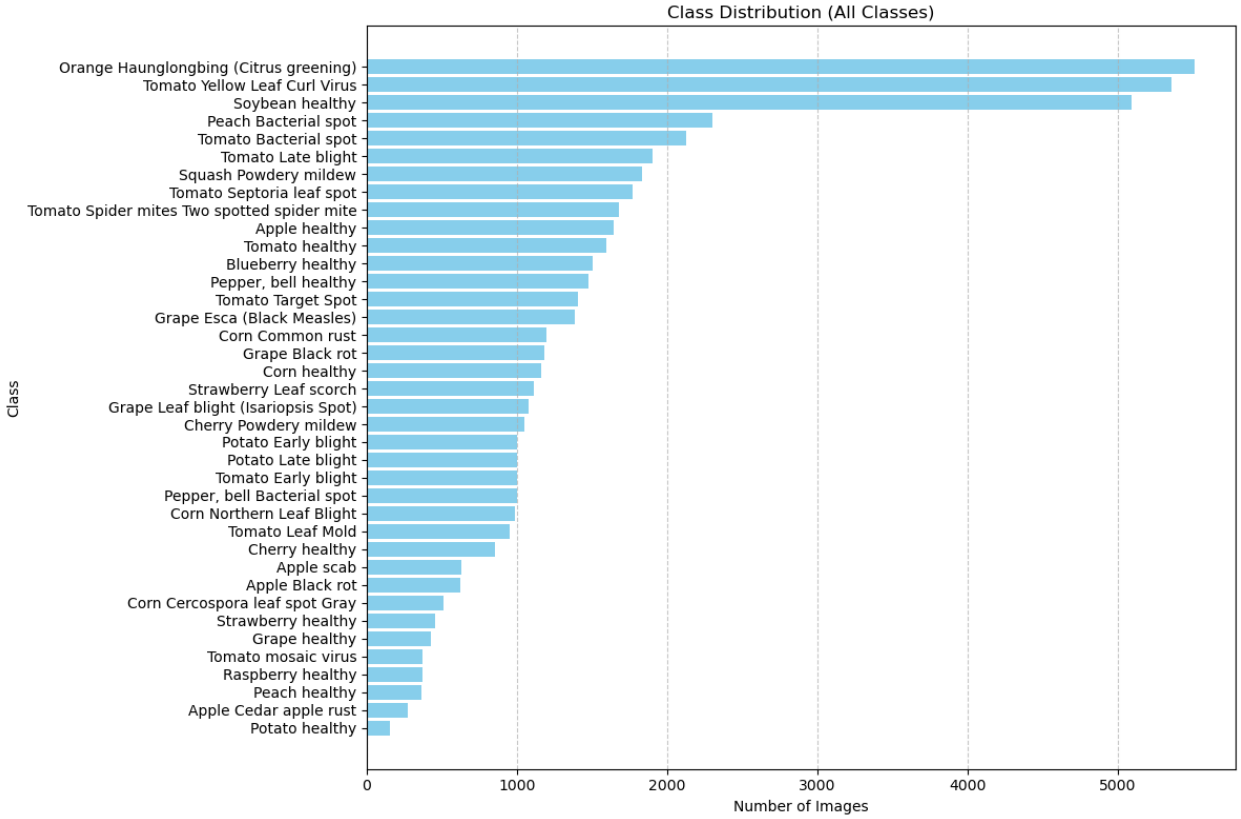
4.1 Univariate Analysis

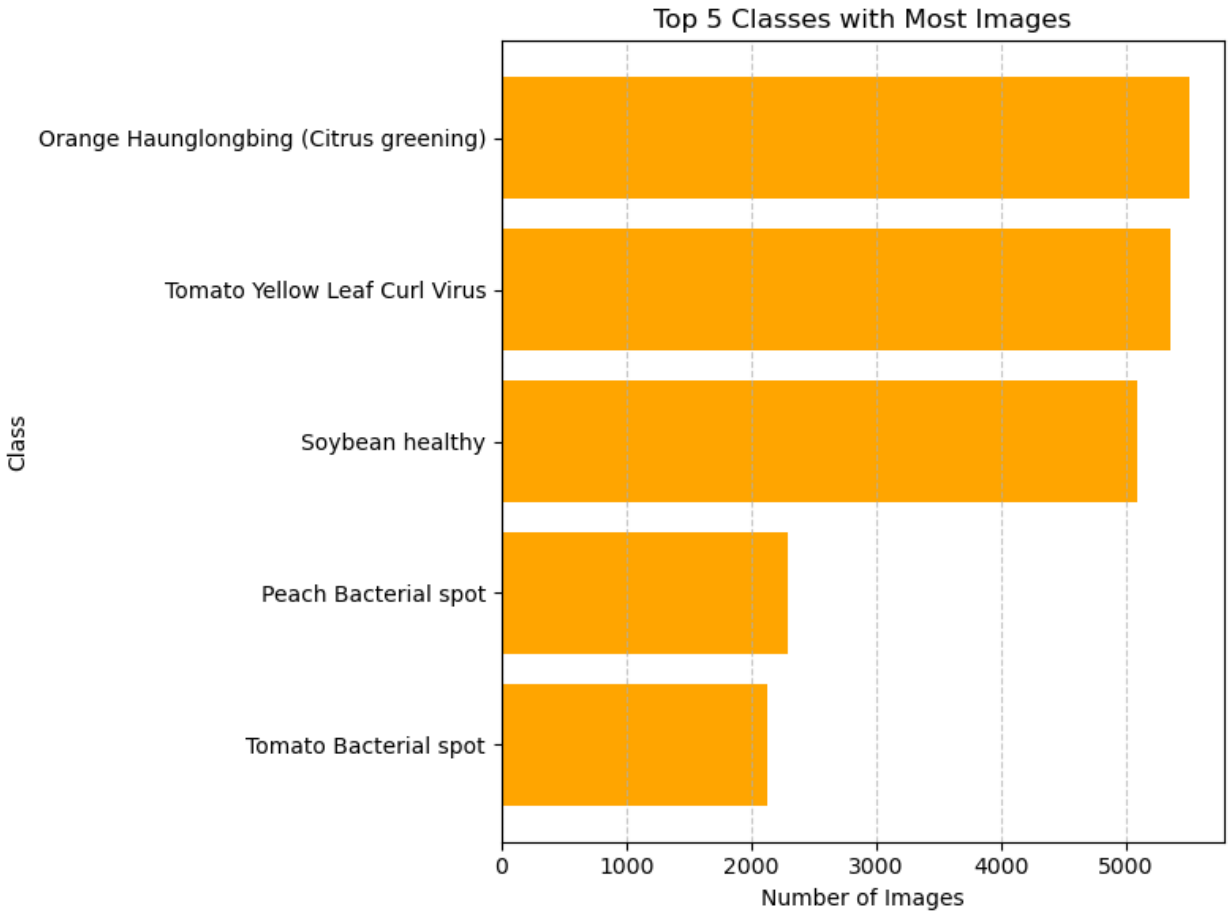
Create a function `plot_class_distribution` that analyzes and visualizes the distribution of images across different classes in the directory.

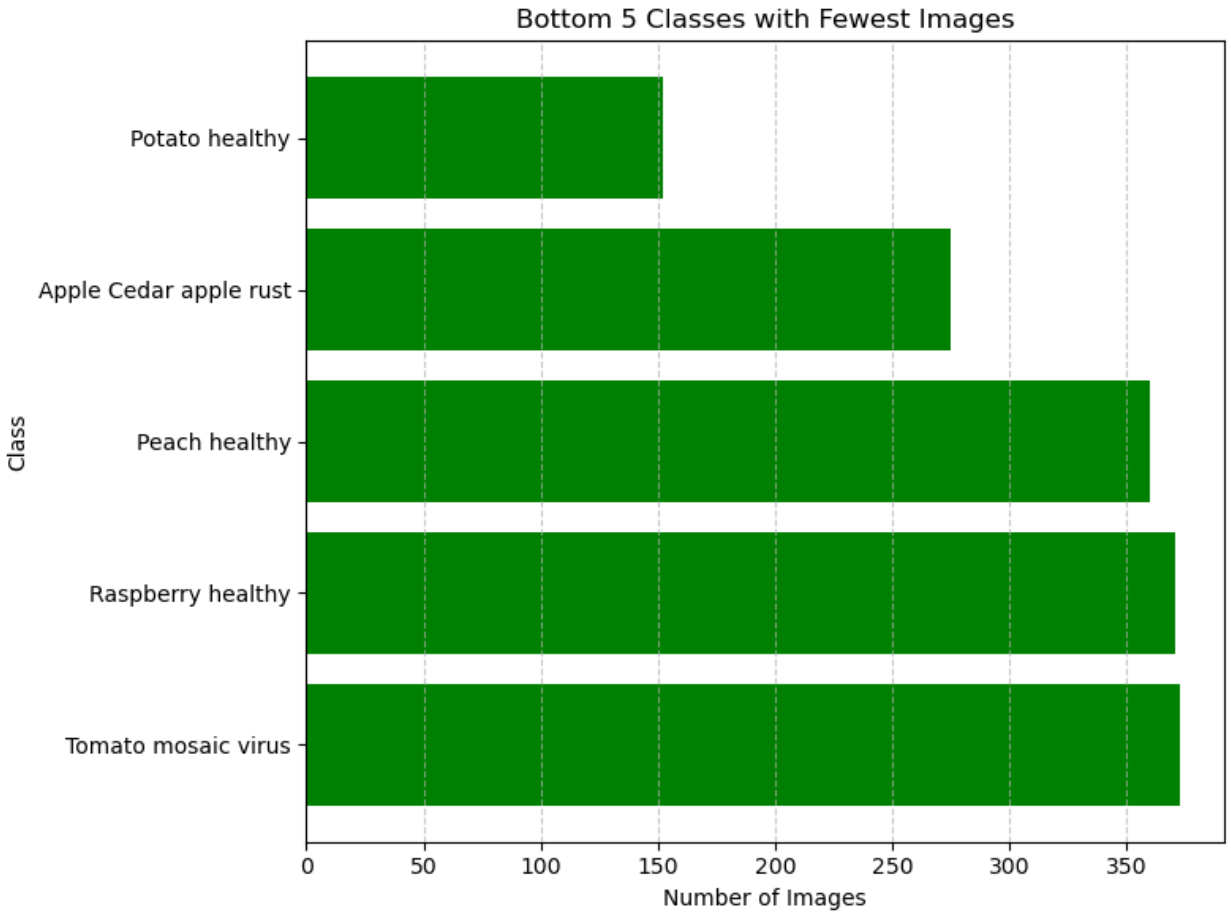
```
# Function to plot class distribution using cleaned class names
def plot_class_distribution(directory):
    class_counts = {}

    # Count the number of images per class
    for root, dirs, files in os.walk(directory):
        if len(files) > 0:
            class_name = os.path.basename(root)
            class_counts[class_name] = len(files)

    # Clean the class names
    original_classes = list(class_counts.keys())
    cleaned_classes = clean_class_names(original_classes)

    # Mapping the original class names to cleaned class names
    cleaned_class_counts = {cleaned: class_counts[original]
                            for original, cleaned in
zip(original_classes, cleaned_classes)}

    # Sorting cleaned class counts in descending order
```

```
    cleaned_class_counts = dict(sorted(cleaned_class_counts.items(),
key=lambda item: item[1], reverse=True))

    # Plotting for all classes as horizontal bar plot
    plt.figure(figsize=(12, 8))
    plt.barh(list(cleaned_class_counts.keys()),
list(cleaned_class_counts.values()), color='skyblue')

    plt.ylabel('Class')
    plt.xlabel('Number of Images')
    plt.title('Class Distribution (All Classes)')

    plt.grid(axis='x', linestyle='--', alpha=0.7)
    plt.gca().invert_yaxis()  # Invert y-axis to show the highest
values at the top
    plt.tight_layout()
    plt.show()

    # Plotting for the top 5 classes as horizontal bar plot (already
in descending order)
    top_5_classes = dict(list(cleaned_class_counts.items())[:5])

    plt.figure(figsize=(8, 6))
    plt.barh(list(top_5_classes.keys()), list(top_5_classes.values()),
color='orange')

    plt.ylabel('Class')
    plt.xlabel('Number of Images')
    plt.title('Top 5 Classes with Most Images')

    plt.grid(axis='x', linestyle='--', alpha=0.7)
    plt.gca().invert_yaxis()  # Invert y-axis to show the highest
values at the top
    plt.tight_layout()
    plt.show()

    # Plotting for the bottom 5 classes as horizontal bar plot (sort
in ascending order)
    bottom_5_classes = dict(sorted(cleaned_class_counts.items(),
key=lambda item: item[1])[:5])

    plt.figure(figsize=(8, 6))
    plt.barh(list(bottom_5_classes.keys()),
list(bottom_5_classes.values()), color='green')

    plt.ylabel('Class')
    plt.xlabel('Number of Images')
    plt.title('Bottom 5 Classes with Fewest Images')
```

```
    plt.grid(axis='x', linestyle='--', alpha=0.7)
    plt.gca().invert_yaxis()  # Invert y-axis to show the highest
values at the top
    plt.tight_layout()
    plt.show()

plot_class_distribution(no_aug_dir)
```




Top 5 Classes with Most Images
Orange Haunglongbing (Citrus greening)
Tomato Yellow Leaf Curl Virus
Soybean healthy
Peach Bacterial spot
Tomato Bacterial spot
Class
0
1000
2000
3000
4000
5000
Number of Images



Observation

- The dataset exhibits a significant class imbalance, with certain classes containing considerably more samples than others.
- Orange___Haunglongbing_(Citrus_greening) has the highest number of images, while several classes have a relatively small number of samples.
- The distribution is skewed towards specific classes, which can impact model performance.
- Classes with fewer samples might be underrepresented, leading to potential biases during training.

Data Preparation

Create a `DataPreparation` class to prepare image data for training the machine learning model.

```
# Create class DataPreparation
class DataPreparation:
    def __init__(self, data_dir, target_size=(224, 224),
batch_size=32):
```

```
        """
        Initialize with dataset directory and parameters for image
processing.

        :param data_dir: Directory where the dataset is stored.
        :param target_size: Target size for resizing images.
        :param batch_size: Batch size for training and validation.
        """
        self.data_dir = data_dir
        self.target_size = target_size
        self.batch_size = batch_size
        self.label_encoder = LabelEncoder()  # Initialize label
encoder

    def encode_labels(self, classes):
        """
        Encode the clean class labels into numerical format.

        :param classes: List of class names.
        :return: Dictionary mapping cleaned class names to numerical
labels.
        """
        # Clean class names
        clean_classes = clean_class_names(classes)
        # Fit and transform labels
        labels = self.label_encoder.fit_transform(clean_classes)
        # Create label mapping
        label_mapping = dict(zip(self.label_encoder.classes_, labels))
        print(f"Label Mapping: {label_mapping}")
        return label_mapping

    def load_data(self, label_mapping):
        """
        Load the data using ImageDataGenerator and create training and
validation sets.

        :param label_mapping: Dictionary mapping cleaned class names
to numerical labels.
        :return: Training and validation data generators.
        """
        datagen = ImageDataGenerator(rescale=1./255,
validation_split=0.2)  # Rescale images and split into
training/validation sets

        # Create training data generator
        train_generator = datagen.flow_from_directory(
            self.data_dir,
            target_size=self.target_size,
            batch_size=self.batch_size,
```

```
            class_mode='categorical',  # Use 'categorical' for multi-class classification
            subset='training'
        )

        # Create validation data generator
        validation_generator = datagen.flow_from_directory(
            self.data_dir,
            target_size=self.target_size,
            batch_size=self.batch_size,
            class_mode='categorical',  # Use 'categorical' for multi-class classification
            subset='validation'
        )

        return train_generator, validation_generator

# Create an instance of DataLoadingAndUnderstanding
data_loader = DataLoadingAndUnderstanding(
    no_aug_dir='Plant_leave_diseases_dataset_without_augmentation',
)
```

```
Found 54300 images belonging to 38 classes.
```

```
# Access the classes from the instance and ensure it's a list
classes = data_loader.classes

# Initialize DataPreparation with the directory and encode labels
data_preparation = DataPreparation(no_aug_dir)

# Pass the classes correctly
label_mapping = data_preparation.encode_labels(classes)
```

```
Label Mapping: {'Apple Black rot': 0, 'Apple Cedar apple rust': 1, 'Apple healthy': 2, 'Apple scab': 3, 'Blueberry healthy': 4, 'Cherry Powdery mildew': 5, 'Cherry healthy': 6, 'Corn Cercospora leaf spot Gray': 7, 'Corn Common rust': 8, 'Corn Northern Leaf Blight': 9, 'Corn healthy': 10, 'Grape Black rot': 11, 'Grape Esca (Black Measles)': 12, 'Grape Leaf blight (Isariopsis Spot)': 13, 'Grape healthy': 14, 'Orange Haunglongbing (Citrus greening)': 15, 'Peach Bacterial spot': 16, 'Peach healthy': 17, 'Pepper, bell Bacterial spot': 18, 'Pepper, bell healthy': 19, 'Potato Early blight': 20, 'Potato Late blight': 21, 'Potato healthy': 22, 'Raspberry healthy': 23, 'Soybean healthy': 24, 'Squash Powdery mildew': 25, 'Strawberry Leaf scorch': 26, 'Strawberry healthy': 27, 'Tomato Bacterial spot': 28, 'Tomato Early blight': 29, 'Tomato Late blight': 30, 'Tomato Leaf Mold': 31, 'Tomato Septoria leaf spot': 32, 'Tomato Spider mites Two spotted spider mite': 33, 'Tomato Target Spot': 34, 'Tomato Yellow Leaf Curl Virus': 35, 'Tomato healthy': 36, 'Tomato mosaic virus': 37}
```

```
train_gen, val_gen = data_preparation.load_data(label_mapping)
```

```
Found 43451 images belonging to 38 classes.
Found 10849 images belonging to 38 classes.
```

```
def verify_class_indices(train_gen, val_gen):
    """
    Verify that the class indices are consistent across training and
validation datasets.

    :param train_gen: Training data generator.
    :param val_gen: Validation data generator.
    """
    # Get class indices from both generators
    train_class_indices = train_gen.class_indices
    val_class_indices = val_gen.class_indices

    # Check if both generators have the same class indices
    if train_class_indices == val_class_indices:
        print("Class indices are consistent across training and
validation datasets.")
    else:
        print("Class indices are not consistent across training and
validation datasets.")
        print(f"Training class indices: {train_class_indices}")
        print(f"Validation class indices: {val_class_indices}")

# Verify class indices
verify_class_indices(train_gen, val_gen)
```

```
Class indices are consistent across training and validation datasets.
```

MODELLING

BASELINE MODEL

Create a class **BaseModel** to `create`, `train` and `evaluate` a Convolutional Neural Network (CNN) for image classification tasks.

```
# Create class BaseModel
class BaseModel:
    def __init__(self, input_shape, num_classes):
        """
        Initialize the model with input shape and number of classes.

        :param input_shape: Shape of the input images.
        :param num_classes: Number of output classes.
        """
        self.input_shape = input_shape
```

```
        self.num_classes = num_classes
        self.model = self.build_model()

    # Build the model with default parameters
    def build_model(self,
                    conv_layers=None,
                    dense_layers=None,
                    dropout_rate=0.5,
                    optimizer=Adam()):
        """
        Build the CNN model architecture.

        :param conv_layers: List of tuples, where each tuple
represents (filters, kernel_size) for a Conv2D layer.
                            Default is [(32, (3, 3)), (64, (3, 3))].
        :param dense_layers: List of integers where each represents
the number of units in a Dense layer.
                             Default is [128].
        :param dropout_rate: Dropout rate for the Dropout layers.
Default is 0.5.
        :param optimizer: Optimizer to compile the model. Default is
Adam.

        :return: Compiled model.
        """
        # Default parameters if none provided
        if conv_layers is None:
            conv_layers = [(32, (3, 3)), (64, (3, 3))]
        if dense_layers is None:
            dense_layers = [128]

        model = Sequential()

        # Add convolutional layers
        for i, (filters, kernel_size) in enumerate(conv_layers):
            if i == 0:
                model.add(Conv2D(filters, kernel_size,
activation='relu', input_shape=self.input_shape))
            else:
                model.add(Conv2D(filters, kernel_size,
activation='relu'))
            model.add(MaxPooling2D((2, 2)))

        # Flatten the output
        model.add(Flatten())

        # Add dense layers
        for units in dense_layers:
            model.add(Dense(units, activation='relu'))
            model.add(Dropout(dropout_rate))
```

```
        # Output layer
        model.add(Dense(self.num_classes, activation='softmax'))

        # Compile the model
        model.compile(optimizer=optimizer,
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

        return model

    def summary(self):
        """Print the summary of the model."""
        self.model.summary()

    def train(self, train_generator, validation_generator, epochs=10):
        """
        Train the model using the provided training and validation
data generators.

        :param train_generator: Training data generator.
        :param validation_generator: Validation data generator.
        :param epochs: Number of epochs to train the model. Default is
10.

        :return: Training history object.
        """
        history = self.model.fit(
            train_generator,
            epochs=epochs,
            validation_data=validation_generator
        )
        return history

# Define input shape and number of classes
input_shape = (224, 224, 3)
num_classes = 38

# Initialize and build the model
model_builder = BaseModel(input_shape, num_classes)

 # Print model summary
model_builder.summary()
```

```
D:\Anaconda\Lib\site-packages\keras\src\layers\convolutional\
base_conv.py:107: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
```

```
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━
┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━
┩
│ conv2d (Conv2D)                 │ (None, 222, 222, 32)   │
896 │
├─────────────────────────────────┼────────────────────────┼───────────────
┤
│ max_pooling2d (MaxPooling2D)    │ (None, 111, 111, 32)   │
0 │
├─────────────────────────────────┼────────────────────────┼───────────────
┤
│ conv2d_1 (Conv2D)               │ (None, 109, 109, 64)   │
18,496 │
├─────────────────────────────────┼────────────────────────┼───────────────
┤
│ max_pooling2d_1 (MaxPooling2D)  │ (None, 54, 54, 64)     │
0 │
├─────────────────────────────────┼────────────────────────┼───────────────
┤
│ flatten (Flatten)               │ (None, 186624)         │
0 │
├─────────────────────────────────┼────────────────────────┼───────────────
┤
│ dense (Dense)                   │ (None, 128)            │
23,888,000 │
├─────────────────────────────────┼────────────────────────┼───────────────
┤
│ dropout (Dropout)               │ (None, 128)            │
0 │
├─────────────────────────────────┼────────────────────────┼───────────────
┤
│ dense_1 (Dense)                 │ (None, 38)             │
4,902 │
└─────────────────────────────────┴────────────────────────┴───────────────
┘

 Total params: 23,912,294 (91.22 MB)

 Trainable params: 23,912,294 (91.22 MB)

 Non-trainable params: 0 (0.00 B)
```

Observation

Conv2D Layers:

- These layers detect features in the images.
- The first one has 32 filters, and the second has 64 filters.

MaxPooling2D Layers:

- These layers reduce the size of the feature maps, making the model more efficient.

Flatten Layer:

- This layer flattens the 3D output from the convolutional layers into a 1D array, preparing it for the dense layers.

Dense Layers:

- These fully connected layers make the final predictions. The first dense layer has 128 neurons, and the last one outputs 38 classes.

Dropout Layer:

- Prevent overfitting by randomly setting some of the neurons to zero during training.

`model_builder` has about `23,912,294` parameters and is designed to process and classify images into 38 categories.

```
# Train the model
history = model_builder.train(train_gen, val_gen, epochs=10)
```

```
Epoch 1/10

D:\Anaconda\Lib\site-packages\keras\src\trainers\data_adapters\
py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should
call `super().__init__(**kwargs)` in its constructor. `**kwargs` can
include `workers`, `use_multiprocessing`, `max_queue_size`. Do not
pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()

1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1229s 902ms/step - accuracy: 0.3712 -
loss: 2.6054 - val_accuracy: 0.7051 - val_loss: 1.0480
Epoch 2/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1324s 974ms/step - accuracy: 0.6449 -
loss: 1.1764 - val_accuracy: 0.8217 - val_loss: 0.6001
Epoch 3/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1346s 990ms/step - accuracy: 0.7211 -
loss: 0.8927 - val_accuracy: 0.8363 - val_loss: 0.5492
Epoch 4/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1294s 953ms/step - accuracy: 0.7697 -
loss: 0.7203 - val_accuracy: 0.8560 - val_loss: 0.4716
Epoch 5/10
```

```
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1231s 906ms/step - accuracy: 0.8083 -
loss: 0.5910 - val_accuracy: 0.8582 - val_loss: 0.4635
Epoch 6/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1254s 923ms/step - accuracy: 0.8347 -
loss: 0.4950 - val_accuracy: 0.8687 - val_loss: 0.4434
Epoch 7/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1223s 900ms/step - accuracy: 0.8554 -
loss: 0.4318 - val_accuracy: 0.8658 - val_loss: 0.4852
Epoch 8/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1200s 883ms/step - accuracy: 0.8726 -
loss: 0.3690 - val_accuracy: 0.8614 - val_loss: 0.5091
Epoch 9/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1215s 894ms/step - accuracy: 0.8811 -
loss: 0.3557 - val_accuracy: 0.8693 - val_loss: 0.4929
Epoch 10/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1211s 891ms/step - accuracy: 0.8956 -
loss: 0.3066 - val_accuracy: 0.8720 - val_loss: 0.4771
```

Evaluating and Saving Model Performance Results**

```
# Save the model in keras format
model_builder.model.save('base_model.keras')
```

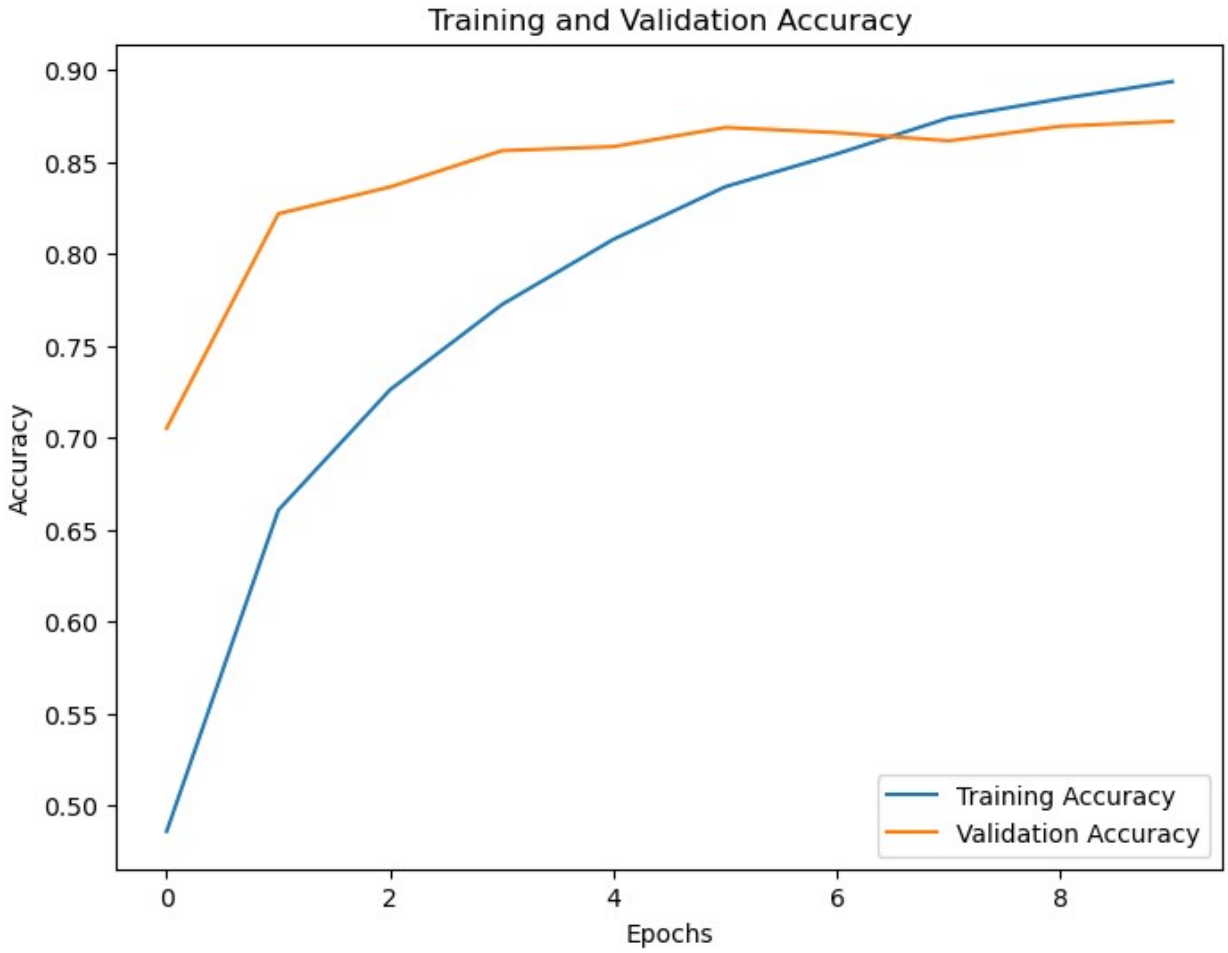
Test the Model

```
# Create a class to test the model
class ModelTester:
    def __init__(self, model_path, class_labels, target_size):
        """
        Initialize with the path to the saved model, class labels, and
target image size.

        :param model_path: Path to the saved model file.
        :param class_labels: Dictionary mapping class indices to class
names.
        :param target_size: Target size for resizing images.
        """
        # Load the trained model
        self.model = load_model(model_path)
        # Class labels mapping
        self.class_labels = class_labels
        self.target_size = target_size

    def preprocess_image(self, img_path):
        """
        Load and preprocess the image for prediction.

        :param img_path: Path to the image file.
        :return: Preprocessed image.
```

```
        """
        img = image.load_img(img_path, target_size=self.target_size)
# Load image
        img_array = image.img_to_array(img)  # Convert image to array
        img_array = np.expand_dims(img_array, axis=0)  # Add batch
dimension
        img_array = img_array / 255.0  # Normalize image
        return img_array

    def predict(self, img_path):
        """
        Predict the class of the image.

        :param img_path: Path to the image file.
        :return: Predicted class label.
        """
        img_array = self.preprocess_image(img_path)  # Preprocess the
image
        predictions = self.model.predict(img_array)  # Make prediction
        predicted_class_index = np.argmax(predictions, axis=1)  # Get
index of the highest probability
        predicted_class_label =
self.class_labels[predicted_class_index[0]]  # Map index to class
label
        return predicted_class_label

# Define model path, image path, and class labels
model_path = 'base_model.keras'  # Path to the saved model file
img_path = 'Grayleaf.png'  # Path to the new image

# Define class labels manually based on the cleaned class names
class_labels = {
    0: 'Apple Apple scab',
    1: 'Apple Black rot',
    2: 'Apple Cedar apple rust',
    3: 'Apple healthy',
    4: 'Background without leaves',
    5: 'Blueberry healthy',
    6: 'Cherry healthy',
    7: 'Cherry Powdery mildew',
    8: 'Corn Cercospora leaf spot Gray leaf spot',
    9: 'Corn Common rust',
    10: 'Corn healthy',
    11: 'Corn Northern Leaf Blight',
    12: 'Grape Black rot',
    13: 'Grape Esca Black Measles',
    14: 'Grape healthy',
    15: 'Grape Leaf blight Isariopsis Leaf Spot',
    16: 'Orange Haunglongbing Citrus greening',
```

```
    17: 'Peach Bacterial spot',
    18: 'Peach healthy',
    19: 'Pepper bell Bacterial spot',
    20: 'Pepper bell healthy',
    21: 'Potato Early blight',
    22: 'Potato healthy',
    23: 'Potato Late blight',
    24: 'Raspberry healthy',
    25: 'Soybean healthy',
    26: 'Squash Powdery mildew',
    27: 'Strawberry healthy',
    28: 'Strawberry Leaf scorch',
    29: 'Tomato Bacterial spot',
    30: 'Tomato Early blight',
    31: 'Tomato Leaf Mold',
    32: 'Tomato Septoria leaf spot',
    33: 'Tomato Spider mites Two-spotted spider mite',
    34: 'Tomato Target Spot',
    35: 'Tomato Tomato Yellow Leaf Curl Virus',
    36: 'Tomato healthy',
    37: 'Tomato Tomato mosaic virus',
    38: 'Tomato Late blight'
}

# Initialize the ModelTester class
model_tester = ModelTester(model_path=model_path,
class_labels=class_labels, target_size=(224, 224))

# Predict the class of the new image
predicted_class = model_tester.predict(img_path)
print(f"The predicted class for the image is: {predicted_class}")

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 274ms/step
The predicted class for the image is: Cherry Powdery mildew

# Extracting accuracy and validation accuracy from the history object
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

# Extracting the number of epochs
epochs_range = range(len(train_accuracy))

# Plotting the graph
plt.figure(figsize=(8, 6))
plt.plot(epochs_range, train_accuracy, label='Training Accuracy')
plt.plot(epochs_range, val_accuracy, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
```

```
plt.legend(loc='lower right')
plt.show()
```



Observation

- Model starts with low accuracy but improves significantly over the epochs.
- Validation accuracy increases steadily at first, suggesting the model is learning well.
- Midway through training, the model begins to stabilize, with slower improvements and some fluctuation in validation accuracy and loss.
- At the end, the model achieves high training accuracy, but there's a slight indication of overfitting, as validation accuracy and loss don't improve as much.

CLASS IMBALANCE

- Next is to deal with class imbalance in the base model to get more accurate results.

```
# Add callbacks and class_weight to the BaseModel
class BaseModel:
    def __init__(self, input_shape, num_classes):
        """
        Initialize the model with input shape and number of classes.
```

```
        :param input_shape: Shape of the input images.
        :param num_classes: Number of output classes.
        """
        self.input_shape = input_shape
        self.num_classes = num_classes
        self.model = self.build_model()  # Build the model with
default parameters

    def build_model(self,
                    conv_layers=None,
                    dense_layers=None,
                    dropout_rate=0.5,
                    optimizer=Adam()):
        """
        Build the CNN model architecture.

        :param conv_layers: List of tuples, where each tuple
represents (filters, kernel_size) for a Conv2D layer.
                            Default is [(32, (3, 3)), (64, (3, 3))].
        :param dense_layers: List of integers where each represents
the number of units in a Dense layer.
                             Default is [128].
        :param dropout_rate: Dropout rate for the Dropout layers.
Default is 0.5.
        :param optimizer: Optimizer to compile the model. Default is
Adam.

        :return: Compiled model.
        """
        # Default parameters if none provided
        if conv_layers is None:
            conv_layers = [(32, (3, 3)), (64, (3, 3))]
        if dense_layers is None:
            dense_layers = [128]

        model = Sequential()

        # Add convolutional layers
        for i, (filters, kernel_size) in enumerate(conv_layers):
            if i == 0:
                model.add(Conv2D(filters, kernel_size,
activation='relu', input_shape=self.input_shape))
            else:
                model.add(Conv2D(filters, kernel_size,
activation='relu'))
            model.add(MaxPooling2D((2, 2)))

        # Flatten the output
        model.add(Flatten())
```

```
        # Add dense layers
        for units in dense_layers:
            model.add(Dense(units, activation='relu'))
            model.add(Dropout(dropout_rate))

        # Output layer
        model.add(Dense(self.num_classes, activation='softmax'))

        # Compile the model
        model.compile(optimizer=optimizer,
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])

        return model

    def summary(self):
        """Print the summary of the model."""
        self.model.summary()

    def train(self, train_generator, validation_generator, epochs=10,
callbacks=None, class_weight=None):
        """
        Train the model using the provided training and validation
data generators.

        :param train_generator: Training data generator.
        :param validation_generator: Validation data generator.
        :param epochs: Number of epochs to train the model. Default is
10.
        :param callbacks: List of callbacks to pass to the fit method.
        :param class_weight: Dictionary mapping class indices to a
weight for the class.

        :return: Training history object.
        """
        history = self.model.fit(
            train_generator,
            epochs=epochs,
            validation_data=validation_generator,

            # Add callbacks also
            callbacks=callbacks,

            # Add class_weight to the fit method
            class_weight=class_weight
        )
        return history
```

```
# Assuming you have the classes and their labels in the training set
y_train = train_gen.classes  # Get the labels from the data generator

# Compute class weights
class_weights = compute_class_weight(class_weight='balanced',
classes=np.unique(y_train), y=y_train)

# Convert the class weights to a dictionary
class_weights = {i: class_weights[i] for i in
range(len(class_weights))}

# Set up EarlyStopping
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

# Train the model with class weights
history_ClassWeights = model_builder.model.fit(
    train_gen,
    epochs=10,
    validation_data=val_gen,
    callbacks=[early_stopping],
    class_weight=class_weights
)
```

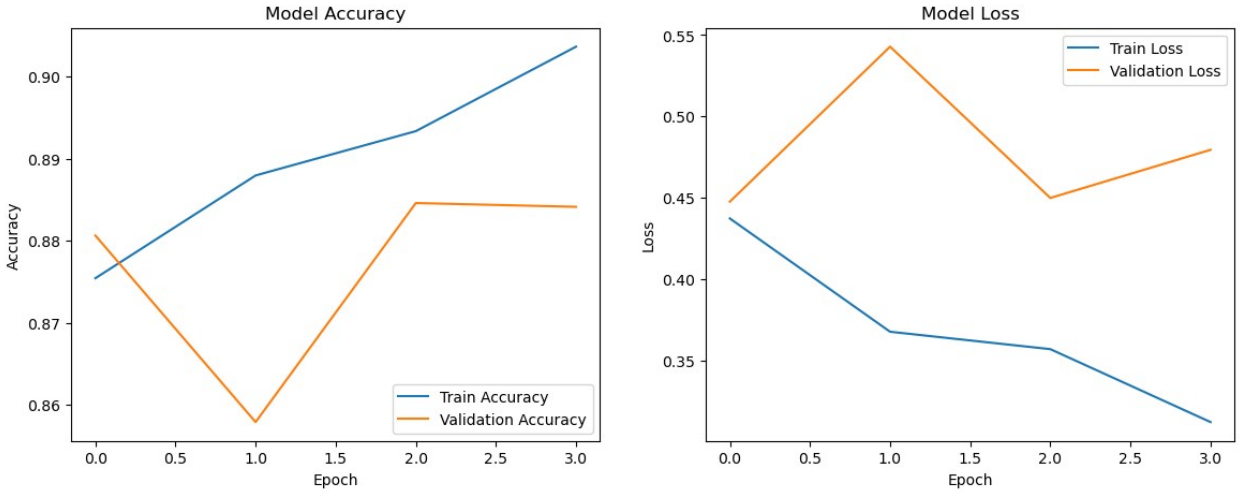
```
Epoch 1/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1207s 886ms/step - accuracy: 0.8774 -
loss: 0.4428 - val_accuracy: 0.8806 - val_loss: 0.4476
Epoch 2/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1207s 888ms/step - accuracy: 0.8903 -
loss: 0.3574 - val_accuracy: 0.8579 - val_loss: 0.5429
Epoch 3/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1202s 884ms/step - accuracy: 0.8977 -
loss: 0.3388 - val_accuracy: 0.8846 - val_loss: 0.4498
Epoch 4/10
1358/1358 ━━━━━━━━━━━━━━━━━━━━ 1287s 947ms/step - accuracy: 0.9056 -
loss: 0.3136 - val_accuracy: 0.8841 - val_loss: 0.4795
```

```
# Save the model
model_builder.model.save('class_imbalance.keras')

# Define model path, image path, and class labels
model_path = 'class_imbalance.keras'  # Path to the saved model file
img_path = 'Grayleaf.png'  # Path to the new image

# Initialize the ModelTester class
model_tester = ModelTester(model_path=model_path,
class_labels=class_labels, target_size=(224, 224))
```

```
# Predict the class of the new image
predicted_class = model_tester.predict(img_path)
print(f"The predicted class for the image is: {predicted_class}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 745ms/step
The predicted class for the image is: Cherry Powdery mildew
```

```
# Plot training & validation accuracy values
plt.figure(figsize=(14, 5))

plt.subplot(1, 2, 1)
plt.plot(history_ClassWeights.history['accuracy'], label='Train
Accuracy')
plt.plot(history_ClassWeights.history['val_accuracy'],
label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')

# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history_ClassWeights.history['loss'], label='Train Loss')
plt.plot(history_ClassWeights.history['val_loss'], label='Validation
Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')

plt.show()
```



Observation

- Model starts with strong performance, indicating it's well-trained from the beginning.

- Validation accuracy fluctuates, with some signs of instability or overfitting as the model progresses.
- Model's overall performance is good, but there are slight challenges with generalization across epochs.

DATA AUGMENTATION

Next is to use data augmentation to artificially expand the size and diversity of the dataset by applying various transformations.

Create a function `get_augmented_data_generators` that sets up data generators to streamline the process of loading and processing image data during model training and evaluation.

```
class DataPreparationForAugmentation:
    def __init__(self, directory, target_size=(224, 224),
batch_size=32, class_mode='categorical',
                 rotation_range=20, width_shift_range=0.2,
height_shift_range=0.2,
                 shear_range=0.2, zoom_range=0.2,
horizontal_flip=True, fill_mode='nearest',
                 val_split=0.2):
        self.directory = directory
        self.target_size = target_size
        self.batch_size = batch_size
        self.class_mode = class_mode
        self.val_split = val_split

        self.rotation_range = rotation_range
        self.width_shift_range = width_shift_range
        self.height_shift_range = height_shift_range
        self.shear_range = shear_range
        self.zoom_range = zoom_range
        self.horizontal_flip = horizontal_flip
        self.fill_mode = fill_mode

    def _get_image_data_generator(self, validation_split):
        datagen = ImageDataGenerator(
            rescale=1./255,
            rotation_range=self.rotation_range,
            width_shift_range=self.width_shift_range,
            height_shift_range=self.height_shift_range,
            shear_range=self.shear_range,
            zoom_range=self.zoom_range,
            horizontal_flip=self.horizontal_flip,
            fill_mode=self.fill_mode,
            validation_split=validation_split
        )
```

```
        # Training data generator
        train_gen = datagen.flow_from_directory(
            self.directory,
            target_size=self.target_size,
            batch_size=self.batch_size,
            class_mode=self.class_mode,
            subset='training'
        )

        # Validation data generator
        val_gen = datagen.flow_from_directory(
            self.directory,
            target_size=self.target_size,
            batch_size=self.batch_size,
            class_mode=self.class_mode,
            subset='validation'
        )

        return train_gen, val_gen

    def load_data(self):
        return
self._get_image_data_generator(validation_split=self.val_split)

# Usage
directory = 'Plant_leave_diseases_dataset_without_augmentation'
data_preparation = DataPreparationForAugmentation(directory)
train_gen, val_gen = data_preparation.load_data()

# Check number of images
print(f"Training images: {train_gen.samples}")
print(f"Validation images: {val_gen.samples}")
```

```
Found 43451 images belonging to 38 classes.
Found 10849 images belonging to 38 classes.
Training images: 43451
Validation images: 10849
```

```
def visualize_augmented_images(train_gen):
    # Get a batch of augmented images
    images, labels = next(train_gen)

    # Plot some images
    plt.figure(figsize=(12, 8))
    for i in range(9):
        plt.subplot(3, 3, i+1)
        plt.imshow(images[i])
        plt.axis('off')
    plt.show()
```

```
# Usage
visualize_augmented_images(train_gen)
```

```
def create_model(input_shape, num_classes):
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu',
input_shape=input_shape),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(128, (3, 3), activation='relu'),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(num_classes, activation='softmax')
    ])
    return model

input_shape = (224, 224, 3)
num_classes = 38
```

```
model = create_model(input_shape, num_classes)
```

```
D:\Anaconda\Lib\site-packages\keras\src\layers\convolutional\
base_conv.py:107: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

```
model.compile(
    optimizer=Adam(),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Early Stopping
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

# Train the model
history_Augmentation = model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=10,
    steps_per_epoch=train_gen.samples // train_gen.batch_size,
    validation_steps=val_gen.samples // val_gen.batch_size,
    callbacks=[early_stopping]
)
```

```
Epoch 1/10
```

```
D:\Anaconda\Lib\site-packages\keras\src\trainers\data_adapters\
py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should
call `super().__init__(**kwargs)` in its constructor. `**kwargs` can
include `workers`, `use_multiprocessing`, `max_queue_size`. Do not
pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
```

```
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 2071s 2s/step - accuracy: 0.2785 -
loss: 2.7782 - val_accuracy: 0.5663 - val_loss: 1.5324
Epoch 2/10
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 1s 151us/step - accuracy: 0.4375 -
loss: 1.7776 - val_accuracy: 1.0000 - val_loss: 1.2896
Epoch 3/10
```

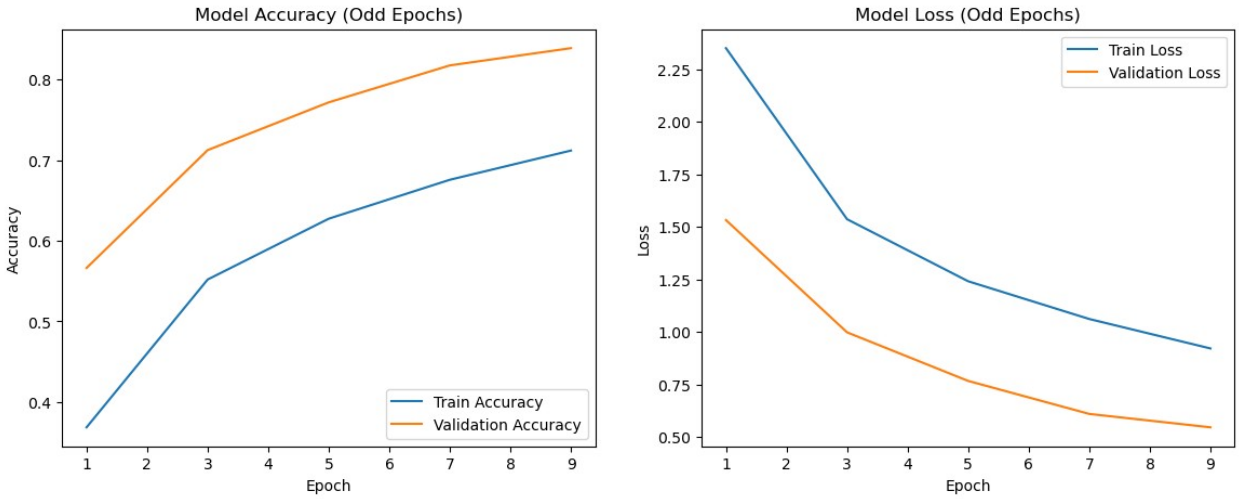
```
D:\Anaconda\Lib\contextlib.py:155: UserWarning: Your input ran out of
data; interrupting training. Make sure that your dataset or generator
can generate at least `steps_per_epoch * epochs` batches. You may need
to use the `.repeat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)

1357/1357 ━━━━━━━━━━━━━━━━━━━━ 2023s 1s/step - accuracy: 0.5265 -
loss: 1.6455 - val_accuracy: 0.7125 - val_loss: 0.9987
Epoch 4/10
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 1s 143us/step - accuracy: 0.4688 -
loss: 1.3910 - val_accuracy: 1.0000 - val_loss: 1.0371
Epoch 5/10
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 2022s 1s/step - accuracy: 0.6130 -
loss: 1.2936 - val_accuracy: 0.7718 - val_loss: 0.7677
Epoch 6/10
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 1s 103us/step - accuracy: 0.7812 -
loss: 0.7914 - val_accuracy: 0.0000e+00 - val_loss: 3.0633
Epoch 7/10
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 1991s 1s/step - accuracy: 0.6672 -
loss: 1.0973 - val_accuracy: 0.8176 - val_loss: 0.6108
Epoch 8/10
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 1s 344us/step - accuracy: 0.8750 -
loss: 0.5170 - val_accuracy: 1.0000 - val_loss: 0.0411
Epoch 9/10
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 1922s 1s/step - accuracy: 0.7031 -
loss: 0.9518 - val_accuracy: 0.8390 - val_loss: 0.5471
Epoch 10/10
1357/1357 ━━━━━━━━━━━━━━━━━━━━ 1s 233us/step - accuracy: 0.8125 -
loss: 0.6564 - val_accuracy: 1.0000 - val_loss: 0.0059

# Save the model
model.save('Augmentation.keras')
```

Define model path, image path, and class labels

```
# Path to the saved model file
model_path = 'Augmentation.keras'

# Path to the new image
img_path = 'test4.jpg'

# Initialize the ModelTester class
model_tester = ModelTester(model_path=model_path,
class_labels=class_labels, target_size=(224, 224))

# Predict the class of the new image
predicted_class = model_tester.predict(img_path)
print(f"The predicted class for the image is: {predicted_class}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 274ms/step
The predicted class for the image is: Corn Cercospora leaf spot Gray
leaf spot
```

```python
# Extract the history data
accuracy = history_Augmentation.history['accuracy']
val_accuracy = history_Augmentation.history['val_accuracy']
loss = history_Augmentation.history['loss']
val_loss = history_Augmentation.history['val_loss']

# Create a list of odd-numbered epochs
odd_epochs = list(range(1, len(accuracy) + 1, 2))

# Filter the accuracy and loss values for odd epochs
odd_accuracy = [accuracy[i - 1] for i in odd_epochs]
odd_val_accuracy = [val_accuracy[i - 1] for i in odd_epochs]
odd_loss = [loss[i - 1] for i in odd_epochs]
odd_val_loss = [val_loss[i - 1] for i in odd_epochs]

# Plot training & validation accuracy values for odd epochs
plt.figure(figsize=(14, 5))

plt.subplot(1, 2, 1)
plt.plot(odd_epochs, odd_accuracy, label='Train Accuracy')
plt.plot(odd_epochs, odd_val_accuracy, label='Validation Accuracy')
plt.title('Model Accuracy (Odd Epochs)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')

# Plot training & validation loss values for odd epochs
plt.subplot(1, 2, 2)
plt.plot(odd_epochs, odd_loss, label='Train Loss')
plt.plot(odd_epochs, odd_val_loss, label='Validation Loss')
plt.title('Model Loss (Odd Epochs)')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')

plt.show()
```



Observation

- Both training and validation accuracy do not show consistent in improvement, indicating that the model is not learning and generalizing well over time.

Recommendation

- Use the `Augmentation.keras`, as it shows consistent performance across multiple metrics and test sets.
- The higher accuracy suggests it might handle class imbalances effectively.