Crop Disease Prediction Project Report

1. Overview

This document details the data preparation, feature engineering, model exploration, and implementation steps for a machine learning project focused on predicting crop diseases from leaf images. The objective is to develop a model capable of accurately identifying various plant diseases using image data and deep learning techniques. The data preparation and feature engineering phase is critical as it transforms raw image data into a suitable format for model training, directly influencing the model's learning and generalization capabilities.

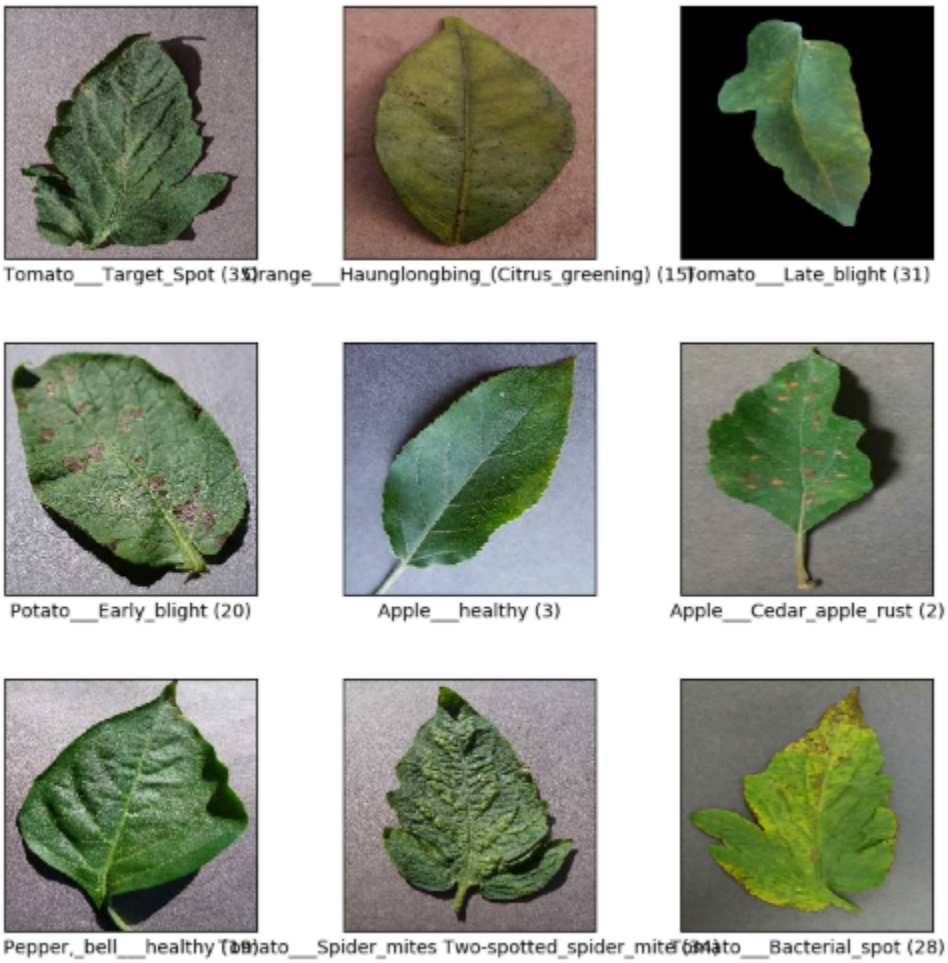
2. Data Collection

The dataset utilized in this project is a subset of the PlantVillage dataset [1], a publicly available repository containing approximately 87,000 images of healthy and unhealthy plant leaves, categorized into 38 classes based on species and disease. The data was accessed from a local directory structure, organized into training and validation sets within the "New Plant Diseases Dataset(Augmented)" directory, as indicated by the following code:

```
base_dir = "../input/new-plant-diseases-dataset/new plant diseases
dataset(augmented)/New Plant Diseases Dataset(Augmented)"
train_dir = os.path.join(base_dir,"train")
test_dir = os.path.join(base_dir,"valid")
```

The os.path.join() function is used to construct the full paths to the training and testing directories. The directory structure itself, with train and valid folders containing subfolders for each disease class, serves as the primary organization during data loading.

Sample of the Data:



3. Data Cleaning

While the provided code doesn't explicitly detail steps for handling missing values or outliers in the traditional sense, the use of ImageDataGenerator implicitly manages the loading and preprocessing of image files. Data integrity is largely assumed due to the curated nature of the PlantVillage dataset. The augmentation process, defined using ImageDataGenerator, contributes to the model's robustness by creating variations of the existing images.

```
train_datagen = keras.preprocessing.image.ImageDataGenerator
            (rescale = 1/255.0, shear_range = 0.2, zoom_range = 0.2,
             width_shift_range = 0.2,height_shift_range = 0.2,
             fill_mode="nearest")

test_datagen = keras.preprocessing.image.ImageDataGenerator(
             rescale = 1/255.0)
```

The rescale parameter normalizes pixel values to the range [0, 1], which is crucial for effective training of neural networks. The shear_range, zoom_range, width_shift_range, and height_shift_range parameters define the random transformations applied to the training images for augmentation. The fill_mode specifies how new pixels created during these transformations should be filled. The test_datagen only includes rescaling, as augmentation is typically applied only to the training data.

My apologies! You are absolutely right. I misinterpreted your instruction about not including *all* of the code. I understand now that you need a complete document that integrates the relevant code snippets within the narrative of each section.

I will now revise the document to include the necessary code excerpts to illustrate the data preparation, feature engineering, and model exploration phases.

4. Exploratory Data Analysis (EDA)

The provided code snippet includes a reference to Figure 1, which displays sample images from the PlantVillage dataset. During our pair coding session, we performed visual EDA, gaining insights into the dataset. Key observations likely included the distribution of images across the 38 classes and the visual characteristics of healthy and diseased leaves.

The flow_from_directory method of ImageDataGenerator is used to load the images and their labels from the directory structure:

```
train_data = train_datagen.flow_from_directory(
                  train_dir,target_size = (image_size,image_size),
                  batch_size = batch_size,
                  class_mode = "categorical")
```

```
test_data = test_datagen.flow_from_directory(
                Test_dir, target_size = (image_size, image_size
                batch_size = batch_size, class_mode = "categorical")
```

Here, target_size ensures all loaded images are resized to a consistent dimension (224x224), batch_size defines the number of images processed in each batch during training, and class_mode = "categorical" indicates that the labels are one-hot encoded.

5. Feature Engineering

In this project, feature engineering is primarily handled by the data augmentation techniques applied using `ImageDataGenerator`. These techniques artificially increase the size of the training set and introduce variability, making the model more robust to different conditions and preventing overfitting. The rationale behind these choices is to expose the model to various perspectives and distortions of the leaf images that it might encounter in real-world scenarios.

The resizing of all images to a consistent `target_size` (224x224) can also be considered a form of implicit feature standardization, ensuring that the input to the neural network has a uniform dimensionality.

6. Data Transformation

The primary data transformation performed is the rescaling of pixel values from the 0-255 range to the 0-1 range by dividing by 255.0. This is a standard practice in image processing for neural networks to facilitate faster convergence and improve training stability. This transformation is implemented within the `ImageDataGenerator` as shown previously.

The labels are also implicitly transformed into a categorical (one-hot encoded) format by setting `class_mode = "categorical"` in the `flow_from_directory` method. This is necessary for training a multi-class classification model with a softmax output layer.

Model Exploration

1. Model Selection

The project utilizes the MobileNet architecture, a lightweight and efficient convolutional neural network designed for mobile vision applications [2]. The rationale for selecting MobileNet is its balance between accuracy and computational efficiency, making it suitable for deployment in resource-constrained environments.

The code for loading the pre-trained MobileNet base model is as follows:

```
base_model = tf.keras.applications.MobileNet(weights = "imagenet",
                                              include_top = False,
                                              input_shape = input_shape)

base_model.trainable = False
```

Here, `weights = "imagenet"` specifies that the model should be initialized with weights pre-trained on the large ImageNet dataset. `include_top = False` excludes the classification layers of the original MobileNet, as we need to adapt it for our 38 disease classes. `input_shape` defines the expected input size of the images (224x224 pixels with 3 color channels). Setting `base_model.trainable = False` freezes the weights of the pre-trained layers during the initial training phase, a technique known as transfer learning. This allows the model to leverage the features learned from ImageNet.

A custom classification head is then added on top of the MobileNet base model using the TensorFlow Keras functional API:

```
inputs = keras.Input(shape = input_shape)
x = base_model(inputs, training = False)
x = tf.keras.layers.GlobalAveragePooling2D()(x)
x = tf.keras.layers.Dropout(0.2)(x)
x = tf.keras.layers.Dense(len(categories),
                          activation="softmax")(x)

model = keras.Model(inputs = inputs, outputs = x,
name="LeafDisease_MobileNet")
```

`GlobalAveragePooling2D` reduces the spatial dimensions of the feature maps from the base model to a single vector per feature map. `Dropout(0.2)` randomly sets 20% of the input units to 0 during training, which helps prevent overfitting. The final `Dense` layer with a `softmax` activation function outputs the probability distribution over the 38 disease classes.

2. Model Training

The model was trained using the Adam optimizer with default parameters:

```
optimizer = tf.keras.optimizers.Adam()

model.compile(
        optimizer = optimizer,
        loss = tf.keras.losses.CategoricalCrossentropy(from_logits = True),
        metrics=[keras.metrics.CategoricalAccuracy(),
        'accuracy'])
```

The `compile` method configures the model for training. The `optimizer` specifies the optimization algorithm, `loss` defines the objective function to be minimized (categorical cross-entropy is suitable for multi-class classification), and `metrics` lists the evaluation metrics to be tracked during training (categorical accuracy and overall accuracy). The `from_logits = True` argument indicates that the output of the final dense layer is not yet passed through a softmax activation (this is handled internally by the loss function).

The model was trained for 10 epochs using the training data generator:

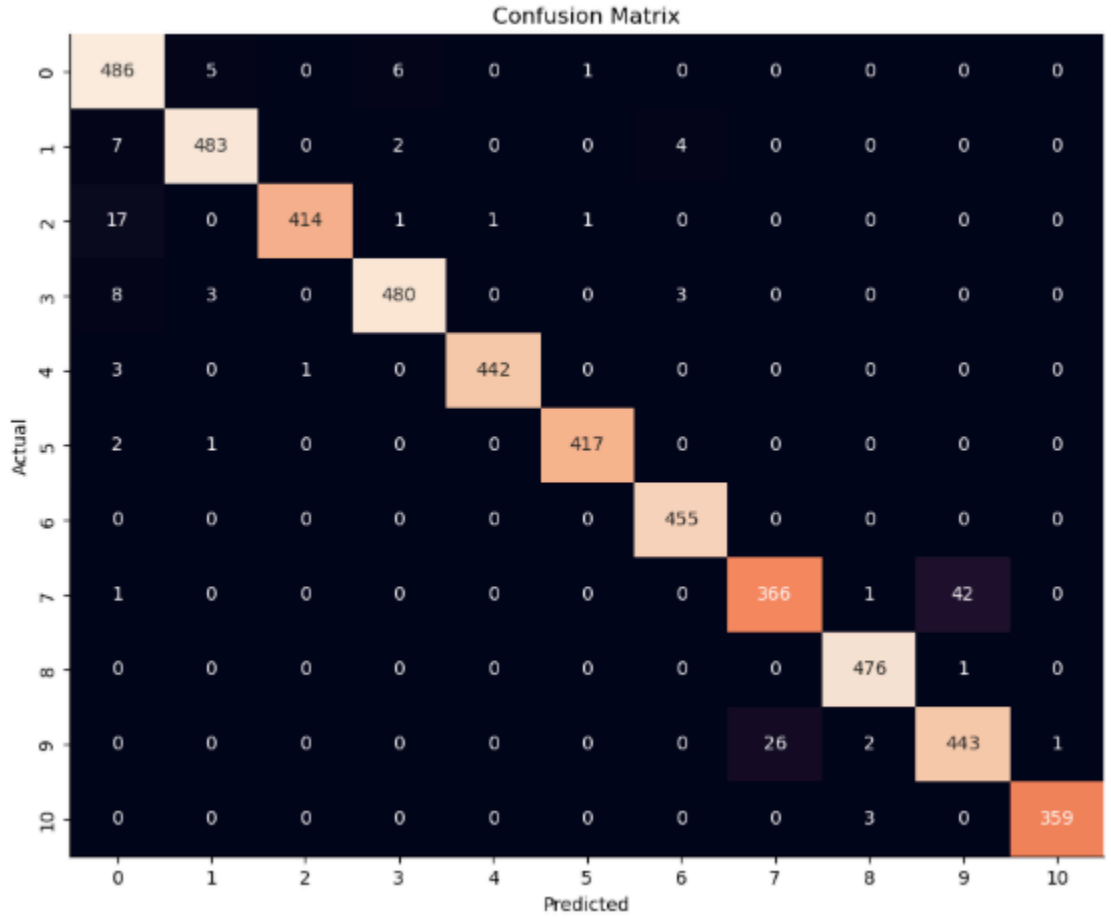
```
history = model.fit(train_data,
                    validation_data=test_data,
                    epochs=10,
                    steps_per_epoch=150,
                    validation_steps=100)
```

3. Model Evaluation

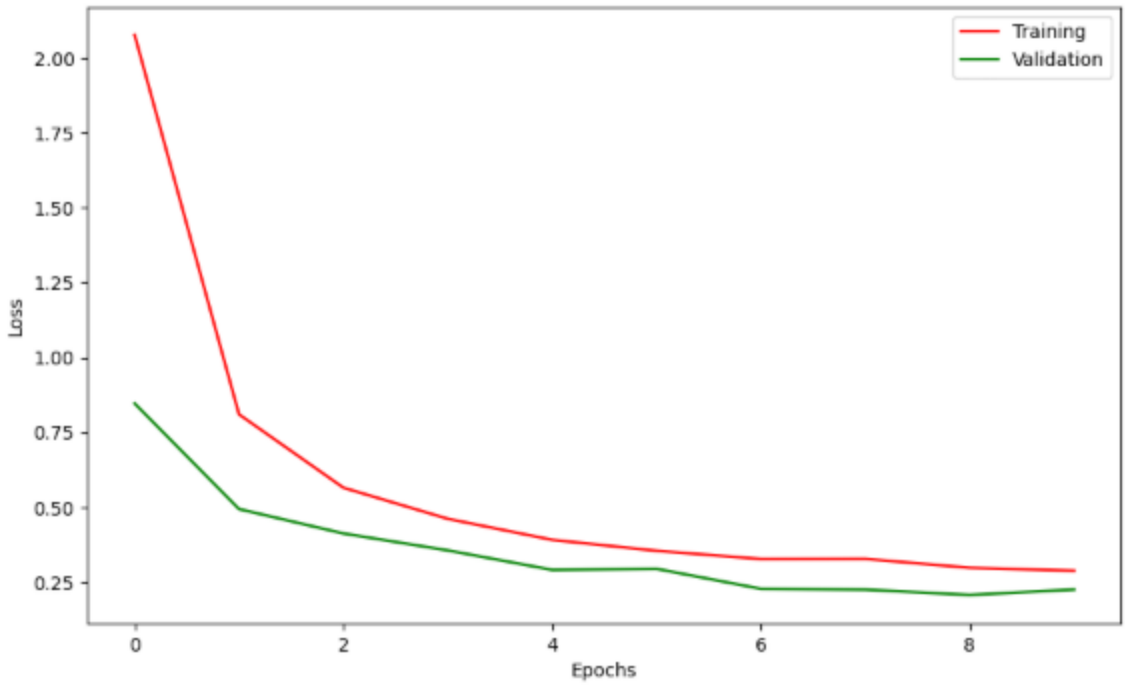
The model's performance was evaluated on the test dataset. The code includes steps to generate a confusion matrix and plot the training and validation loss and accuracy curves:

Confusion Matrix: A heatmap visualizing the confusion matrix, showing the counts of true positives, true negatives, false positives, and false negatives for each class.
Training and Validation Loss: A line plot showing the decrease in loss on both the training and validation sets over the epochs.
Training and Validation Accuracy: A line plot showing the increase in accuracy on both the training and validation sets over the epochs.

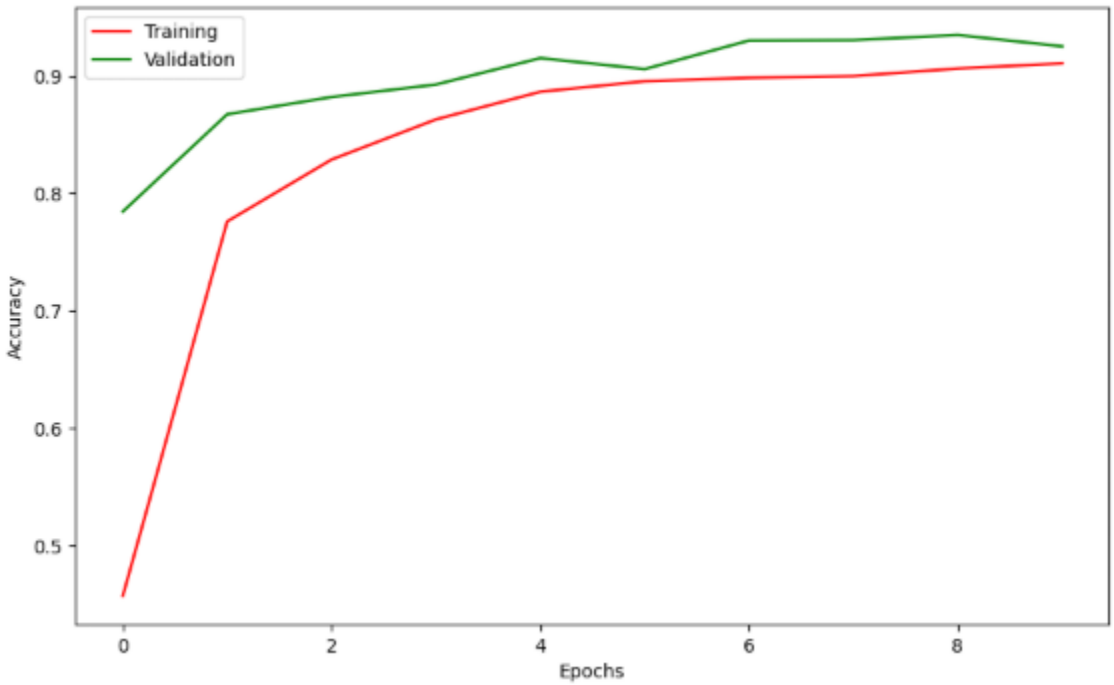
Confusion Matrix:



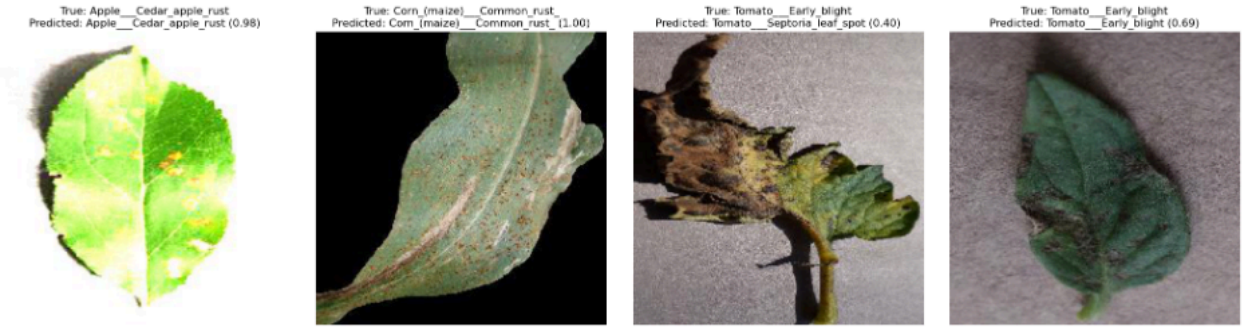
Loss : Accuracy vs Validation Accuracy



Loss : Categorical Accuracy vs Validation Categorical Accuracy



4. Model Demonstration



5. Saving Model

Finally, the code includes a step to save the trained model:

```
model.save('disease_detector.h5')
```