UNIVERSITY OF GUJRAT

A WORLD CLASS UNIVERSITY

PROJECT

Plant Disease Recognition

Machine Learning
(CS-447)

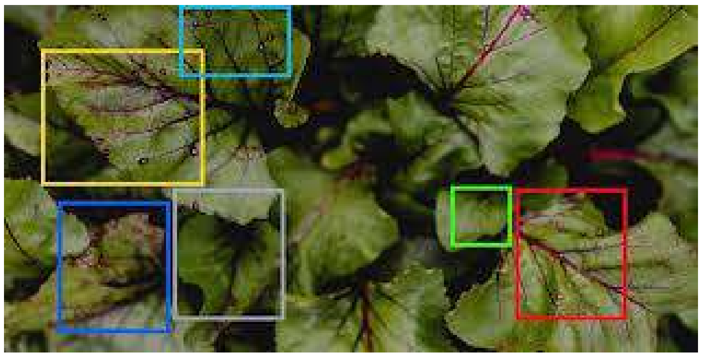
Submitted to: **Dr. Zahid Iqbal**

Anas Majeed [21011519-019]
Enrolled in: BS CS (C)

Chapter 1: Project Proposal

Problem Statement

The aim of this project is to develop a robust and efficient model for plant disease recognition using image data. Plant diseases significantly impact agricultural productivity, making accurate and timely identification crucial for maintaining crop health and yield. This project will leverage convolutional neural networks (CNNs), support vector machines (SVMs), k-nearest neighbors (KNN), and decision trees (DT) to classify plant conditions based on leaf images. The comparison will involve evaluating the performance of these models on two distinct datasets, focusing on their accuracy and generalizability.



Datasets

We will use two different datasets to ensure a comprehensive evaluation of the models:

1. **Plant Disease Recognition Dataset:**

Source: https://www.kaggle.com/datasets/rashikrahmanpritom/plant-disease-recognition-dataset

Description: This dataset contains 1,530 images categorized into train, test, and validation sets. The images are labeled as "Healthy," "Rust," and "Powdery" to describe the plant conditions.

Classes:

- Rust: Caused by Pucciniales fungi, leading to severe plant deformities.
- Powdery: Caused by Erysiphales fungi, reducing crop yields.
- Healthy: Plants without any diseases.

2. **PlantVillage Dataset:**

Source: https://www.kaggle.com/datasets/emmarex/plantdisease

Description: This dataset comprises 4,000 images of healthy and infected leaves of crop plants, curated from the PlantVillage platform. This dataset consists of about 4K rgb images of healthy and diseased crop leaves which is categorized into different classes. The total dataset is divided into 80/20 ratio of training and validation set preserving the directory structure.

Classifiers

We will apply the following classifiers to both datasets:

1. ***Convolutional Neural Network (CNN):***

CNNs are the backbone of most modern image recognition tasks due to their ability to automatically and adaptively learn spatial hierarchies of features through backpropagation. This makes CNNs particularly well-suited for recognizing plant diseases from leaf images.

2. ***Support Vector Machine (SVM):***
   SVMs are effective in high-dimensional spaces and are still competitive when the number of dimensions exceeds the number of samples. They are particularly effective in cases where the number of samples for training is limited, making them a good candidate for comparison.
3. ***K-Nearest Neighbors (KNN):***
   KNN is a simple, instance-based learning algorithm that can be effective for small datasets. It classifies a data point based on how its neighbors are classified, making it a good method to compare against more complex algorithms.
4. ***Decision Tree (DT):***
   Decision Trees are intuitive and easy to interpret models that split the data into branches to make decisions. They can be useful to see how a non-parametric model performs compared to more sophisticated approaches.

Comparison and Evaluation

The performance of the classifiers will be compared based on several metrics and parameters:

1. ***Accuracy:*** Overall classification accuracy on the test set.
2. ***Precision, Recall, F1-Score:*** Class-wise performance metrics.
3. ***Confusion Matrix:*** To visualize the performance of the classifiers on different classes.
4. ***Training and Inference Time***: Computational efficiency.
5. ***Parameter Tuning:***
   - For CNNs: Different optimization functions (e.g., Adam, SGD), number of layers, activation functions, etc.
   - For SVMs: Different kernels (e.g., linear, RBF), regularization parameters, etc.
   - For KNN: Different values of k (number of neighbors).
   - For DT: Different depth levels and criteria for splitting (e.g., Gini impurity, entropy).

Visualization and Documentation

The results will be presented using various charts and tables:

1. ***Accuracy and Loss Curves:*** To show the training process of CNNs.
2. ***Confusion Matrices:*** For visual comparison of classifier performance.
3. ***Bar Charts:*** To compare accuracy, precision, recall, and F1-score across classifiers and datasets.
4. ***Parameter Sensitivity Analysis:*** Line plots to show the impact of different parameter values on model performance.

All these visualizations and detailed comparisons will be documented in a Jupyter Notebook and a comprehensive project report. This will facilitate understanding and communicating the effectiveness of the classifiers and the impact of different parameters on their performance.

ANAS MAJEED (21011519-019)

BS CS VI - C

Plant Disease Identification using CNN

Table of Contents

- Introduction
    - Convolutional Layer
    - Pooling Layer
    - Fully-Connected Layer
- CNNS and Computer Vision
- This Notebook
- Importing Libraries
- Configuring Notebook
- Exploring the Data
- Preprocessing
- Data Augmentation
- Building the Convolutional Neural Network
- Validating Performance

- Conclusion

Introduction

Convolutional Neural Networks (CNNs or ConvNets) are specialized neural architectures that is predominantly used for several **computer vision** tasks, such as image classification and object recognition. These neural networks harness the power of *Linear Algebra*, specifically through convolution operations, to identify patterns within images.

Convolutional neural networks have three main kinds of layers, which are:

• Convolutional layer

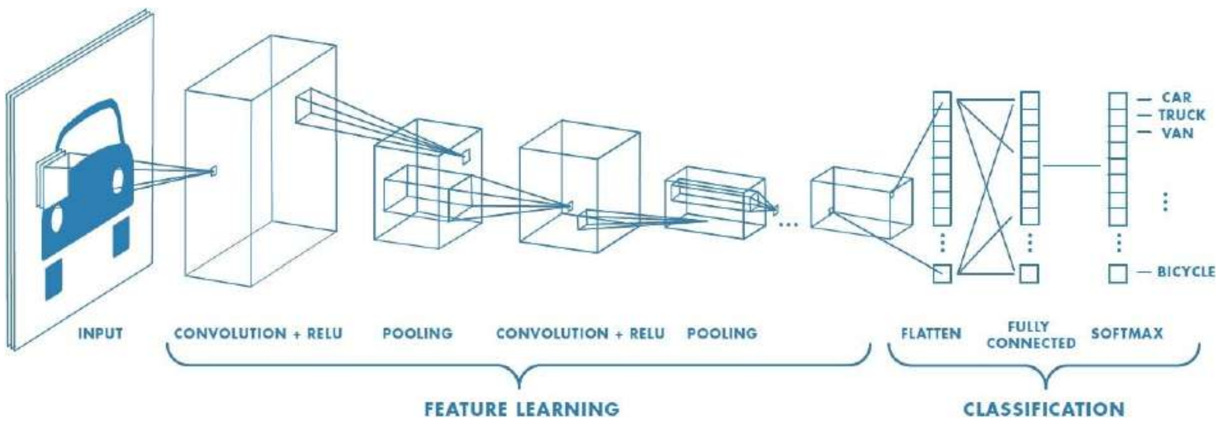
• Pooling layer

• Fully-connected layer

The convolutional layer is the first layer of the network, while the fully-connected layer is the final layer, responsible for the output. The first convolutional layer may be followed by several additional convolutional layers or pooling layers; and with each new layer, the more complex is the CNN.

As the CNN gets more complex, the more it excels in identifying greater portions of the image. Whereas earlier layers focus on the simple features, such as colors and edges; as the image progresses through the network, the CNN starts to recognize larger elements and shapes, until finally reaching its main goal.

The image below displays the structure of a CNN. We have an input image, followed by Convolutional and Pooling layers, where the feature learning process happens. Later on, we have the layers responsible for the task of classifying whether the vehicle in the input data is a car, truck, van, bicycle, etc.



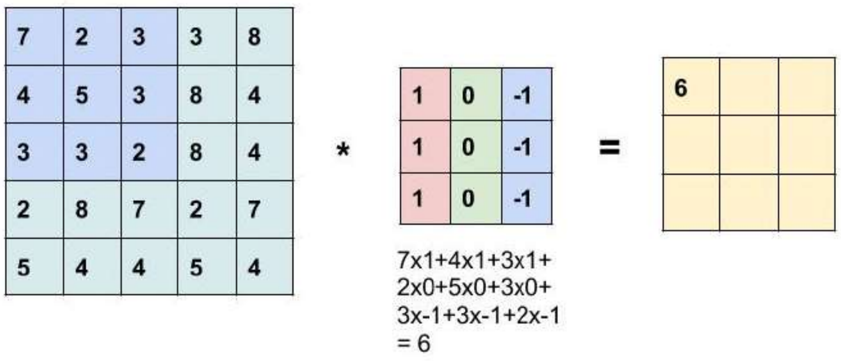
Convolutional Layer

The convolutional layer is the most important layer of a CNN; responsible for dealing with the major computations. The convolutional layer includes **input data, a filter, and a feature map.**

To illustrate how it works, let's assume we have a color image as input. This image is made up of a matrix of pixels in 3D, representing the three dimensions of the image: height, width, and depth.

The filter—which is also referred to as kernel—is a two-dimensional array of weights, and is typically a 3×3 matrix. It is applied to a specific area of the image, and a **dot product is computed between the input pixels and the weights in the filter. Subsequently, the filter shifts by a stride, and this whole process is repeated until the kernel slides through the entire image, resulting in an output array.**

The resulting output array is also known as a feature map, activation map, or convolved feature.



GIF displaying the convolutional process. First, we have a 5×5 matrix—pixels in the input image—with a 3×3 filter. The result of the operation is the output array.
Source: Convolutional Neural Networks

It is important to note that the weights in the filter remain fixed as it moves across the image. The weights values are adjusted during the training process due to backpropagation and gradient descent.

Besides the weights in the filter, we have other three important parameters that need to be set before the training begins:

• **Number of Filters:** This parameter is responsible for defining the depth of the output. If we have three distinct filters, we have three different feature maps, creating a depth of three.

• **Stride:** This is the distance, or number of pixels, that the filter moves over the input matrix.

• **Zero-padding:** This parameter is usually used when the filters do not fit the input image. This sets all elements outside the input matrix to zero, producing a larger or equally sized output. There are three different kinds of padding:

■ **Valid padding:** Also known as no padding. In this specific case, the last convolution is dropped if the dimensions do not align.

■ **Same padding:** This padding ensures that the output layer has the exact same size as the input layer.

■ **Full padding:** This kind of padding increases the size of the output by adding zeros to the borders of the input matrix.

After each convolution operation, we have the application of a **Rectified Linear Unit (ReLU)** function, which transforms the feature map and introduces nonlinearity.

No description has been provided for this image

ReLU activation function:
$f(u)$ = $\begin{cases} 0 & \text{if } u \leq 0\\ u & \text{if } u > 0 \end{cases}$

Source: ResearchGate

As mentioned earlier, the initial convolutional layer can be followed by additional convolutional layers.

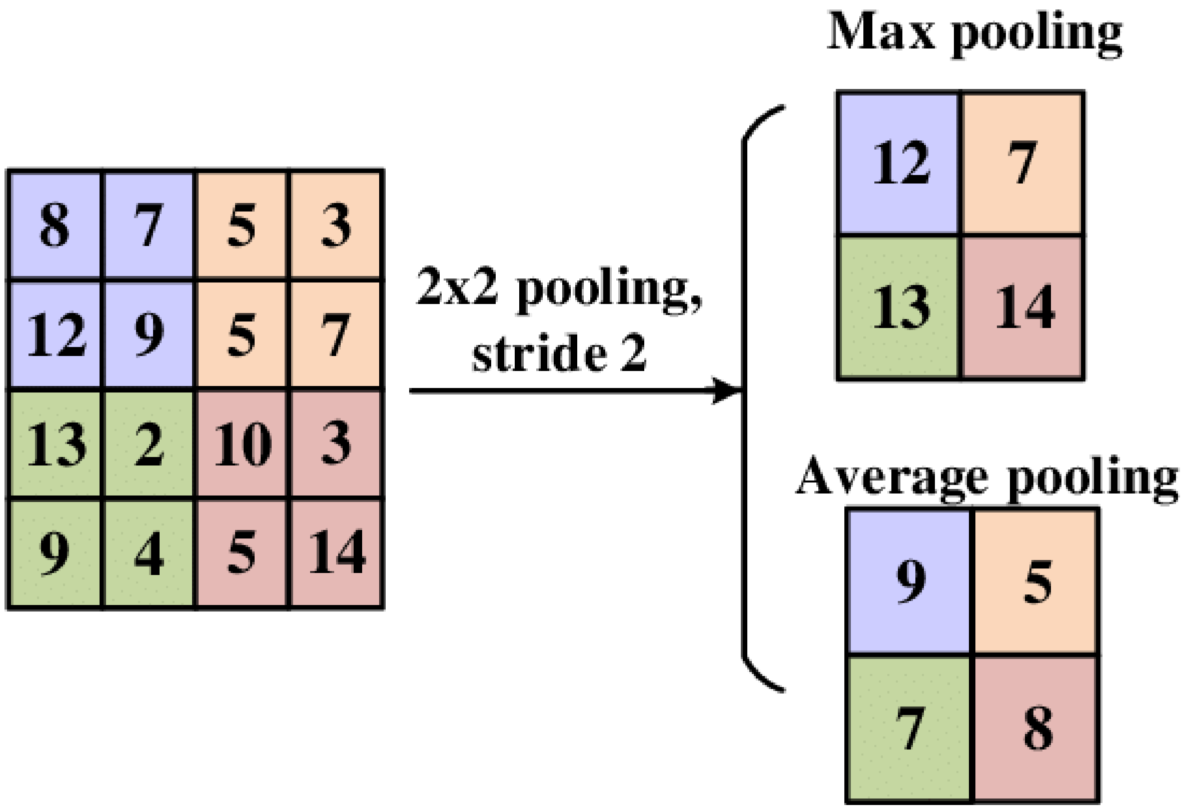
The subsequent convolutional layers can see the pixels within the receptive fields of the prior layers, which helps to extract and interpret additional patterns.

Pooling Layer

The pooling layer is responsible for reducing the dimensionality of the input. It also slides a filter across the entire input—without any weights—to populate the output array. We have two main types of pooling:

• **Max Pooling:** As the filter slides through the input, it selects the pixel with the highest value for the output array.

• **Average Pooling:** The value selected for the output is obtained by computing the average within the receptive field.



The pooling layer serves the purpose of reducing complexity, improving efficiency, and limiting the risk of overfitting.

Fully-Connected Layer

This is the layer responsible for performing the task classification based on the features extracted during the previous layers. While both convolutional and pooling layers tend to use $ReLU$ functions, fully-connected layers use the Softmax activation function for classification, producing a probability from 0 to 1.

No description has been provided for this image

Softmax activation function graph.
Source: ResearchGate

\begin{equation} \sigma(z_i) = \frac{e^{z_{i}}}{\sum_{j=1}^K e^{z_{j}}} \ \ \ \end{equation}

Where:

•

$\sigma{(z_i)}$ = The softmax function applied to the i^{th} element of the input vector. This value ranges between 0 and 1.

•

e^{z_i} = The exponential function applied to the i^{th} element of the input vector.

•

$\sum_{j=1}^K e^{z_{j}}$ = The sum of the exponential of each element in the input vector from to K, where K is the total number of classes/labels.

CNN and Computer Visions

Due to its power in image recognition tasks, CNNs have been highly effective in many fields related to Computer Vision.

Computer Vision is a field of AI that enables computers to extract information from digital images, videos, and other visual inputs. Some common applications of computer vision today can be seen across several industries, including the following:

• **Social Media:** Google, Meta, and Apple use these systems to identify people in a photograph, making it easier to organize photo albums and tag friends.

• **Healthcare:** Computer vision models have been used to help doctors identifying cancerous tumors in patients, as well as other conditions.

• **Agriculture:** Drones equipped with cameras can monitor the health of vast farmlands to identify areas that need more water or fertilizers.

• **Security:** Surveillance systems can detect unusual and suspect activities in real time.

• **Finance:** Computer vision models may be used to identify relevant patterns in candlestick charts to predict price movements.

• **Automotive:** Computer vision is an essential component of the research leading to self-driving cars.

This Notebook

Nowadays, there are several pre-trained CNNs available for many tasks. Models like ***ResNet, VGG16, InceptionV3***, as well as many others, are highly efficient in most computer vision tasks we currently perform across industries.

In this notebook, however, I would like to explore the process of building a simple, yet effective, Convolutional Neural Network from scratch. For this task, I will use **Keras** to help us build a neural network that can accurately identify diseases in a plant through images.

I am going to use the Plant Disease Recognition Dataset, which contains 1,530 images divided into train, test, and validation sets. The images are labeled as “Healthy“, “Rust“, and “Powdery“ to describe the conditions of the plants.

Very briefly, each class means the following:

• **Rust:** These are plant diseases caused by Pucciniales fungi, which cause severe deformities to the plant.

• **Powdery:** Powdery mildews are caused by Erysphales fungi, posing a threat to agriculture and horticulture by reducing crop yields.

• **Healthy:** Naturally, these are the plants that are free from diseases.

Importing Libraries

In [42]:

```
# Importing Libraries

# Data Handling
import pandas as pd
import numpy as np
from collections import defaultdict
from concurrent.futures import ThreadPoolExecutor

# Efficient Looping
import itertools

# Traceback for diagnosis
import traceback

# Data Visualization
import plotly.express as px
import plotly.graph_objs as go
import plotly.subplots as sp
from plotly.subplots import make_subplots
import plotly.figure_factory as ff
import plotly.io as pio
from IPython.display import display
from plotly.offline import init_notebook_mode
init_notebook_mode(connected=True)
from PIL import Image
import matplotlib.pyplot as plt
from concurrent.futures import ThreadPoolExecutor

# Statistics & Mathematics
import scipy.stats as stats
import statsmodels.api as sm
from scipy.stats import shapiro, skew, anderson, kstest
import math

# Feature Selection
from sklearn.feature_selection import (
    RFECV, SelectKBest, chi2, f_classif, f_regression,
    mutual_info_classif, mutual_info_regression
)

# Machine Learning Pipeline
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.base import BaseEstimator, TransformerMixin,ClassifierMi

# Preprocessing data
from sklearn.preprocessing import RobustScaler, StandardScaler, Quant

from sklearn.compose import ColumnTransformer
from sklearn.base import BaseEstimator, TransformerMixin

# Model Selection for Cross Validation
from sklearn.model_selection import (
    StratifiedKFold, KFold,
    RepeatedKFold, RepeatedStratifiedKFold,
    train_test_split, TimeSeriesSplit
)

# Machine Learning metrics
from sklearn.metrics import (
    mean_squared_error,
    r2_score,
    mean_absolute_error,
    cohen_kappa_score,
    make_scorer,
    roc_curve,
    auc,
    accuracy_score,
    f1_score,
    precision_score,
    recall_score,
    confusion_matrix
)

# ML regressors
from sklearn.linear_model import HuberRegressor,RANSACRegressor, Thei
from sklearn.svm import SVR, NuSVR, LinearSVR
from sklearn.ensemble import (
    HistGradientBoostingRegressor, StackingRegressor,
    AdaBoostRegressor, RandomForestRegressor, ExtraTreesRegressor,
    GradientBoostingRegressor, StackingRegressor, VotingRegressor
    )
```

```
# ML classifiers
from sklearn.linear_model import LogisticRegression, RidgeClassifier
from sklearn.svm import SVC, NuSVC, LinearSVC
from sklearn.ensemble import (
    HistGradientBoostingClassifier, AdaBoostClassifier,
    RandomForestClassifier, GradientBoostingClassifier,
    StackingClassifier, VotingClassifier,ExtraTreesClassifier
    )
from sklearn.tree import DecisionTreeClassifier


# Clustering algorithms
from sklearn.cluster import KMeans

# Randomizer
import random

# Encoder of categorical variables
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder

# OS
import os

# Image package
from PIL import Image

# Hiding warnings
import warnings
warnings.filterwarnings("ignore")
```

Importing KERAS

In [ ]:

```
# Importing Keras
from keras.models import Sequential                            # Neural
from keras.layers import Conv2D                                # Convol
from keras.layers import MaxPooling2D                          # Max po
from keras.layers import Flatten                               # Layer
from keras.layers import Dense                                 # This l
from keras.layers import Dropout                               # This s
from keras.layers import BatchNormalization                    # This i
from keras.layers import Activation                            # Layer
from keras.callbacks import EarlyStopping, ModelCheckpoint     # Classe
from keras.models import load_model                            # This h
```

```
# Preprocessing layers
from keras.layers import Rescaling                                          # This l

# Importing TensorFlow
import tensorflow as tf
```

Configuring the Notebook

In [43]:

```
seed = 123
paper_color = '#EEF6FF'
bg_color = '#EEF6FF'

def image_resizer(paths):
    """
    This function resizes the input images
    """
    with ThreadPoolExecutor() as executor:
        resized_images = list(executor.map(lambda x: Image.open(x).re
    return resized_images

# Function to plot images
def plot_images_list(images, title, subtitle):
    '''
    This function helps to plot a matrix of images in a list
    '''
    fig, axes = plt.subplots(3, 3, figsize=(15, 15))
    fig.suptitle(f"{title}\n{subtitle}", fontsize=16)
    images = image_resizer(images)

    for ax, img in zip(axes.flat, images):
        ax.imshow(img)
        ax.axis('off')

    plt.show()
```

In [ ]:

```
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        tf.config.experimental.set_visible_devices(gpus[0], 'GPU')
        strategy = tf.distribute.OneDeviceStrategy(device="/gpu:0")
        print('\nGPU Found! Using GPU...')
    except RuntimeError as e:
        print(e)
        strategy = tf.distribute.get_strategy()
else:
    strategy = tf.distribute.get_strategy()
    print('Number of replicas:', strategy.num_replicas_in_sync)
```

```
GPU Found! Using GPU...
```

Exploring the Data

Before building our Convolutional Neural Network, it is helpful to perform a brief, yet efficient, analysis of the data we have at hand. Let's start by loading the directories for each set.

In [ ]:

```
# Loading training, testing, and validation directories
train_dir = '/content/drive/MyDrive/Dataset 2/Train/Train'
test_dir = '/content/drive/MyDrive/Dataset 2/Test/Test'
val_dir = '/content/drive/MyDrive/Dataset 2/Validation/Validation'
```

We may also count the files inside each subfolder to compute the total of data we have for training and testing, as well as measure the degree of class imbalance.

In [ ]:

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

```
Mounted at /content/drive
```

In [ ]:

```
# Giving names to each directory
directories = {
    train_dir: 'Train',
    test_dir: 'Test',
    val_dir: 'Validation'
    }

# Naming subfolders
subfolders = ['Healthy', 'Powdery', 'Rust']

print('\n* * * * * Number of files in each folder * * * * *\n')
```

```python
# Counting the total of pictures inside each subfolder and directory
for dir, name in directories.items():
    total = 0
    for sub in subfolders:
        path = os.path.join(dir, sub)
        num_files = len([f for f in os.listdir(path) if os.path.join(
        total += num_files
        print(f'\n{name}/{sub}: {num_files}')
    print(f'\n  Total: {total}')
    print("-" * 80)
```

```
* * * * * Number of files in each folder * * * * *


Train/Healthy: 458

Train/Powdery: 430

Train/Rust: 434

  Total: 1322
---------------------------------------------------------------------
-----------

Test/Healthy: 50

Test/Powdery: 50

Test/Rust: 50

  Total: 150
---------------------------------------------------------------------
-----------

Validation/Healthy: 20

Validation/Powdery: 20

Validation/Rust: 20

  Total: 60
---------------------------------------------------------------------
-----------
```

We have a total of **1,322 files** inside the ***Train*** directory and there are no large imbalances between classes. A small variation between them is fine, and a simple metric such as Accuracy may be enough to measure performance.

For the testing set, we have a total of **150** images, whereas the validation set consists of **60** images in total. Both sets have a perfect class balance.

Convolutional Neural Networks require a fixed size for all images we feed into it. This means that every single image in our dataset must be equally sized, either $128 \times 128$, $224 \times 224$, and so on.

We can also check if our data meets this requirement, or if it will be necessary to perform some preprocessing in this regard before modeling.

In [ ]:

```python
unique_dimensions = set()

for dir, name in directories.items():
    for sub in subfolders:
        folder_path = os.path.join(dir, sub)

        for file in os.listdir(folder_path):
            image_path = os.path.join(folder_path, file)
            with Image.open(image_path) as img:
                unique_dimensions.add(img.size)

if len(unique_dimensions) == 1:
    print(f"\nAll images have the same dimensions: {unique_dimensions
else:
    print(f"\nFound {len(unique_dimensions)} unique image dimensions:
```

```
Found 8 unique image dimensions: {(4032, 3024), (4000, 2672), (4000,
3000), (5184, 3456), (2592, 1728), (3901, 2607), (4608, 3456), (2421,
2279)}
```

We have 8 different dimensions across the dataset. In the next cell, I am going to check the distribution of these dimensions across the data.

In [ ]:

```python
# Checking if all the images in the dataset have the same dimensions
dims_counts = defaultdict(int)

for dir, name in directories.items():
    for sub in subfolders:
        folder_path = os.path.join(dir, sub)
        for file in os.listdir(folder_path):
            image_path = os.path.join(folder_path, file)
            with Image.open(image_path) as img:
                dims_counts[img.size] += 1

for dimension, count in dims_counts.items():
    print(f"\nDimension {dimension}: {count} images")
```

```
Dimension (4000, 2672): 1130 images

Dimension (4000, 3000): 88 images

Dimension (2421, 2279): 1 images

Dimension (5184, 3456): 97 images

Dimension (2592, 1728): 127 images

Dimension (4608, 3456): 72 images

Dimension (4032, 3024): 16 images

Dimension (3901, 2607): 1 images
```

It seems that most images have dimensions of $4000 \times 2672$, which is a **rectangular shape**. We can conclude that, due to the differences in dimensions, we will need to apply some preprocessing to the data.

First, we are going to resize the images, so they all have the same shape. Then, we will transform the input from rectangular shape to **square shape**.

Another crucial consideration is verifying the pixel valye range of the images. In this case, all images should have pixel values spanning from **0 to 255**. This consistency simplifies the preprocessing step, since we often normalize pixel values in images to a range going from **0 to 1**.

In [ ]:

```
# Checking images dtype
all_uint8 = True
all_in_range = True

for dir, name in directories.items():
    for sub in subfolders:
        folder_path = os.path.join(dir, sub)
        for file in os.listdir(folder_path):
            image_path = os.path.join(folder_path, file)
            with Image.open(image_path) as img:
                img_array = np.array(img)

            if img_array.dtype == 'uint8':
                all_uint8 = False

            if img_array.min() < 0 or img_array.max() > 255:
                all_in_range = False

if all_uint8:
    print(" - All images are of data type uint8\n")
else:
    print(" - Not all images are of data type uint8\n")

if all_in_range:
    print(" - All images have pixel values ranging from 0 to 255")
else:
    print(" - Not all images have the same pixel values from 0 to 255
```

```
 - Not all images are of data type uint8

 - All images have pixel values ranging from 0 to 255
```

Even though not all images are of the same data type, `uint8`, it is fairly easy to guarantee that they will have the same data type once we load images into datasets. We confirmed, though, that all the images have pixel values ranging from 0 to 255, which is great news.

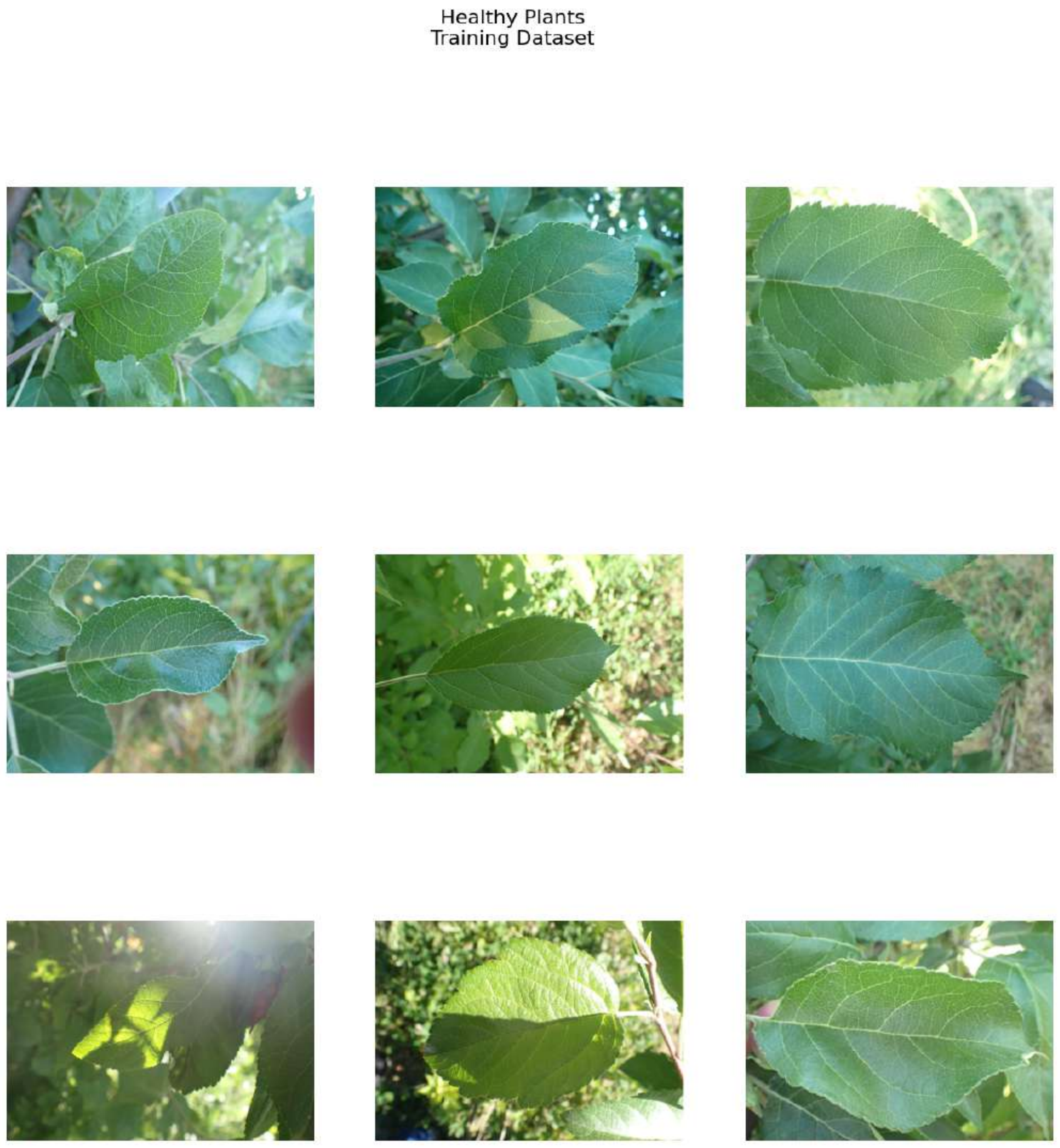
Before moving on to the *Preprocessing step*, let's plot some images from each class to see what they look like.

In [44]:

```
# Loading the directory for each class in the training dataset
train_healthy_dir = train_dir + "/" + 'Healthy'
train_rust_dir = train_dir + "/" + 'Rust'
train_powdery_dir = train_dir + "/" + 'Powdery'

# Selecting 9 random pictures from each directory
healthy_files = random.sample(os.listdir(train_healthy_dir), 9)
rust_files = random.sample(os.listdir(train_rust_dir), 9)
powdery_files = random.sample(os.listdir(train_powdery_dir), 9)
```

In [45]:

```python
# Plotting healthy plants
healthy_images = [os.path.join(train_healthy_dir, f) for f in healthy_
plot_images_list(healthy_images, "Healthy Plants", "Training Dataset"
```

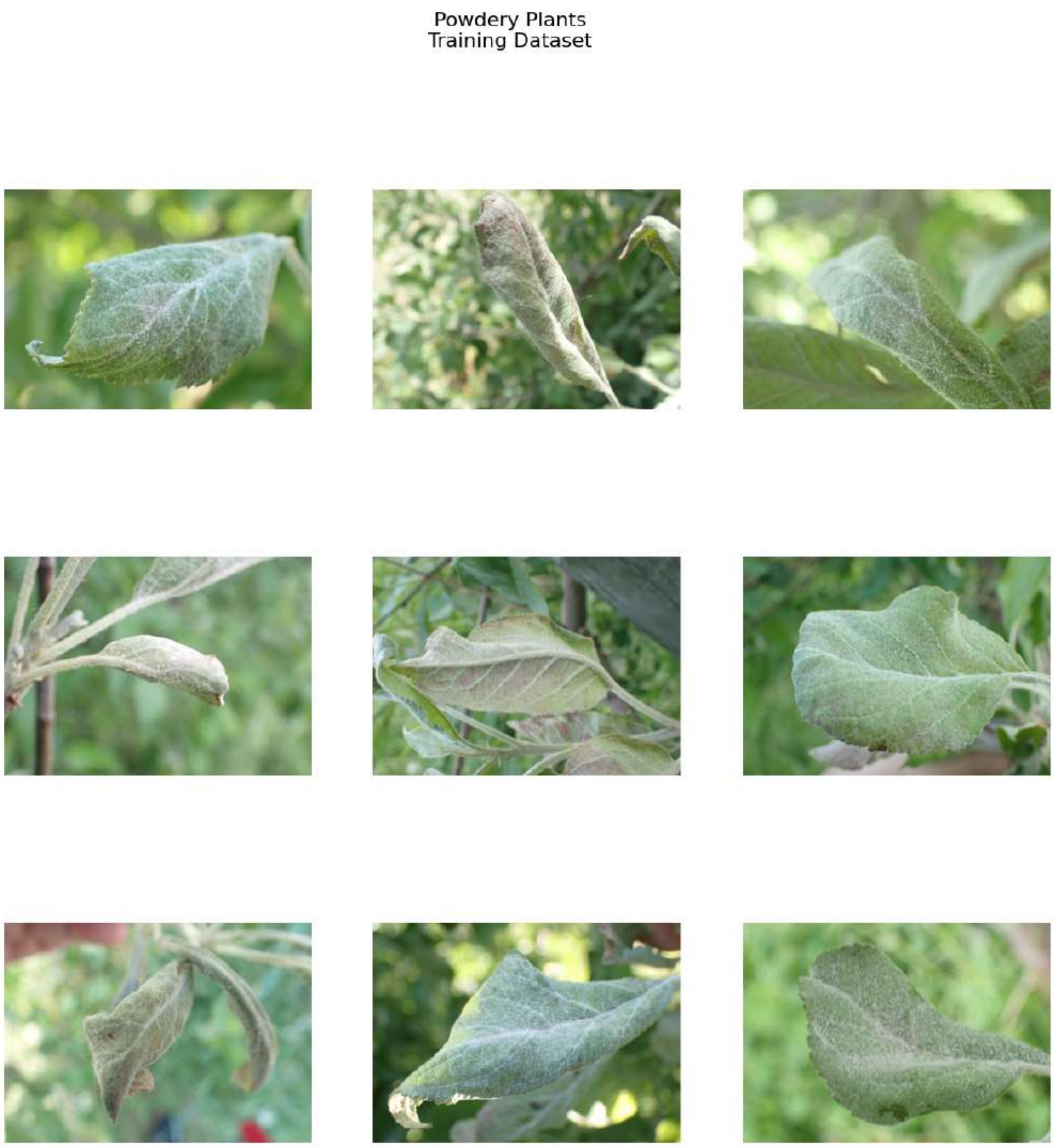


In [46]:

```python
# Plotting rust plants
rust_images = [os.path.join(train_rust_dir, f) for f in rust_files]
plot_images_list(rust_images, "Rust Plants", "Training Dataset")
```

In [47]:

```
# Plotting powdery plants
powdery_images = [os.path.join(train_powdery_dir, f) for f in powdery_
plot_images_list(powdery_images, "Powdery Plants", "Training Dataset"
```



Preprocessing

For those familiar with tabular data, preprocessing is probably one of the most daunting steps of dealing with neural networks and unstructured data.

This task can be fairly easy by using TensorFlow's `image_dataset_from_directory`, which loads images from the directories as a **TensorFlow Dataset**. This resulting dataset can be manipulated for batching, shuffling, augmentating, and several other preprocessing steps.

I suggest you check this link for more information on the `image_dataset_from_directory` function.

In [ ]:

```
# Creating a Dataset for the Training data
train = tf.keras.utils.image_dataset_from_directory(
    train_dir,  # Directory where the Training images are located
    labels = 'inferred', # Classes will be inferred according to the
    label_mode = 'categorical',
    class_names = ['Healthy', 'Powdery', 'Rust'],
    batch_size = 16,    # Number of processed samples before updating
    image_size = (256, 256), # Defining a fixed dimension for all imag
    shuffle = True,  # Shuffling data
    seed = seed,  # Random seed for shuffling and transformations
    validation_split = 0, # We don't need to create a validation set
    crop_to_aspect_ratio = True # Resize images without aspect ratio
)
```

```
Found 1322 files belonging to 3 classes.
```

In [ ]:

```
# Creating a dataset for the Test data
test = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    labels = 'inferred',
    label_mode = 'categorical',
    class_names = ['Healthy', 'Powdery', 'Rust'],
    batch_size = 16,
    image_size = (256, 256),
    shuffle = True,
    seed = seed,
    validation_split = 0,
    crop_to_aspect_ratio = True
)
```

```
Found 150 files belonging to 3 classes.
```

In [ ]:

```
# Creating a dataset for the Test data
validation = tf.keras.utils.image_dataset_from_directory(
    val_dir,
    labels = 'inferred',
    label_mode = 'categorical',
    class_names = ['Healthy', 'Powdery', 'Rust'],
    batch_size = 16,
    image_size = (256, 256),
    shuffle = True,
    seed = seed,
    validation_split = 0,
    crop_to_aspect_ratio = True
)
```

```
Found 60 files belonging to 3 classes.
```

We have successfully captured all files within each set for each of the three classes. We can also print these datasets for a further understanding of their structure.

In [ ]:

```
print('\nTraining Dataset:', train)
print('\nTesting Dataset:', test)
print('\nValidation Dataset:', validation)
```

```
Training Dataset: <_PrefetchDataset element_spec=(TensorSpec(shape=(N
one, 256, 256, 3), dtype=tf.float32, name=None), TensorSpec(shape=(No
ne, 3), dtype=tf.float32, name=None))>

Testing Dataset: <_PrefetchDataset element_spec=(TensorSpec(shape=(No
ne, 256, 256, 3), dtype=tf.float32, name=None), TensorSpec(shape=(Non
e, 3), dtype=tf.float32, name=None))>

Validation Dataset: <_PrefetchDataset element_spec=(TensorSpec(shape=
(None, 256, 256, 3), dtype=tf.float32, name=None), TensorSpec(shape=
(None, 3), dtype=tf.float32, name=None))>
```

Let's explore a bit deeper what all the information above means.

- **_BatchDataset:** It indicates that the dataset returns data in batches.

- **element_spec:** This describes the structure of the elements in the dataset.

- **TensorSpec(shape=(None, 256, 256, 3), dtype=tf.float32, name = None):** This represents the features, in this case the images, in the dataset. `None` represents the batch size, which is *None* here because it can vary depending on how many samples we have in the last batch; `256, 256` represents the height and width of the images; `3` is the number of channels in the images, indicating they are RGB images. Last, `dtype=tf.float32` tells us that the data type of the image pixels is a 32-bit floating point.

- **TensorSpec(shape=(None, 3), dtype=tf.float32, name=None):** This represents the labels/targets of our dataset. Here, `None` refers to the batch size; `3` refers to the number of labels in the dataset; whilst `dtype=tf.float32` is also a 32-bit floating point. By using the `image_dataset_from_directory` function, we have been able to automatically preprocess some aspects of the data. For instance, all the images are now of the same data type, `tf.float32`. By setting `image_size = (256, 256)`, we have ensured that all images have the same dimensions, $256 \times 256$.

Another important step for preprocessing is ensuring that the pixel values of our images are within a 0 to 1 range. The `image_dataset_from_directory` method performed some transformations already, but the pixel values are still in the 0 to 255 range.

In [ ]:

```
# Checking minimum and maximum pixel values in the Validation dataset
min_value = float('inf')
max_value = -float('inf')

for img, label in validation:
    batch_min = tf.reduce_min(img)
    batch_max = tf.reduce_max(img)

    min_value = min(min_value, batch_min.numpy())
    max_value = max(max_value, batch_max.numpy())

print('\nMinimum pixel value in the Validation dataset', min_value)
print('\nMaximum pixel value in the Validation dataset', max_value)
```

```
Minimum pixel value in the Validation dataset 0.0

Maximum pixel value in the Validation dataset 255.0
```

To bring the pixel values to the 0 to 1 range, we can easily use one of Keras' preprocessing layers, `tf.keras.layers.Rescaling`.

In [ ]:

```
scaler = Rescaling(1./255) # Defining scaler values between 0 to 1
```

In [ ]:

```
# Rescaling datasets
```

```
train = train.map(lambda x, y: (scaler(x), y))
test = test.map(lambda x, y: (scaler(x), y))
validation = validation.map(lambda x, y: (scaler(x), y))
```

Now we can once more visualize the minimum and maximum pixel values in the validation set.

In [ ]:

```
# Checking minimum and maximum pixel values in the Validation dataset
min_value = float('inf')
max_value = -float('inf')

for img, label in validation:
    batch_min = tf.reduce_min(img)
    batch_max = tf.reduce_max(img)

    min_value = min(min_value, batch_min.numpy())
    max_value = max(max_value, batch_max.numpy())

print('\nMinimum pixel value in the Validation dataset', min_value)
print('\nMaximum pixel value in the Validation dataset', max_value)
```

```
Minimum pixel value in the Validation dataset 0.0

Maximum pixel value in the Validation dataset 1.0
```

- **tf.keras.layers.RandomCrop**: This layer randomly chooses a location to crop images down to a target size.

- tf.keras.layers.RandomFlip: This layer randomly flips images horizontally and or vertically based on the `mode` attribute.

- **tf.keras.layers.RandomTranslation**: This layer randomly applies translations to each image during training according to the `fill_mode` attribute.

- **tf.keras.layers.RandomBrightness**: This layer randomly increases/reduces the brightness for the input RGB images.

- **tf.keras.layers.RandomRotation**: This layer randomly rotates the images during training, and also fills empty spaces according to the `fill_mode` attribute.

- **tf.keras.layers.RandomZoom**: This layer randomly zooms in or out on each axis of each image independently during training.

- tf.keras.layers.RandomContrast: This layer randomly adjusts contrast by a random factor during training in or out on each axis of each image independently during training.

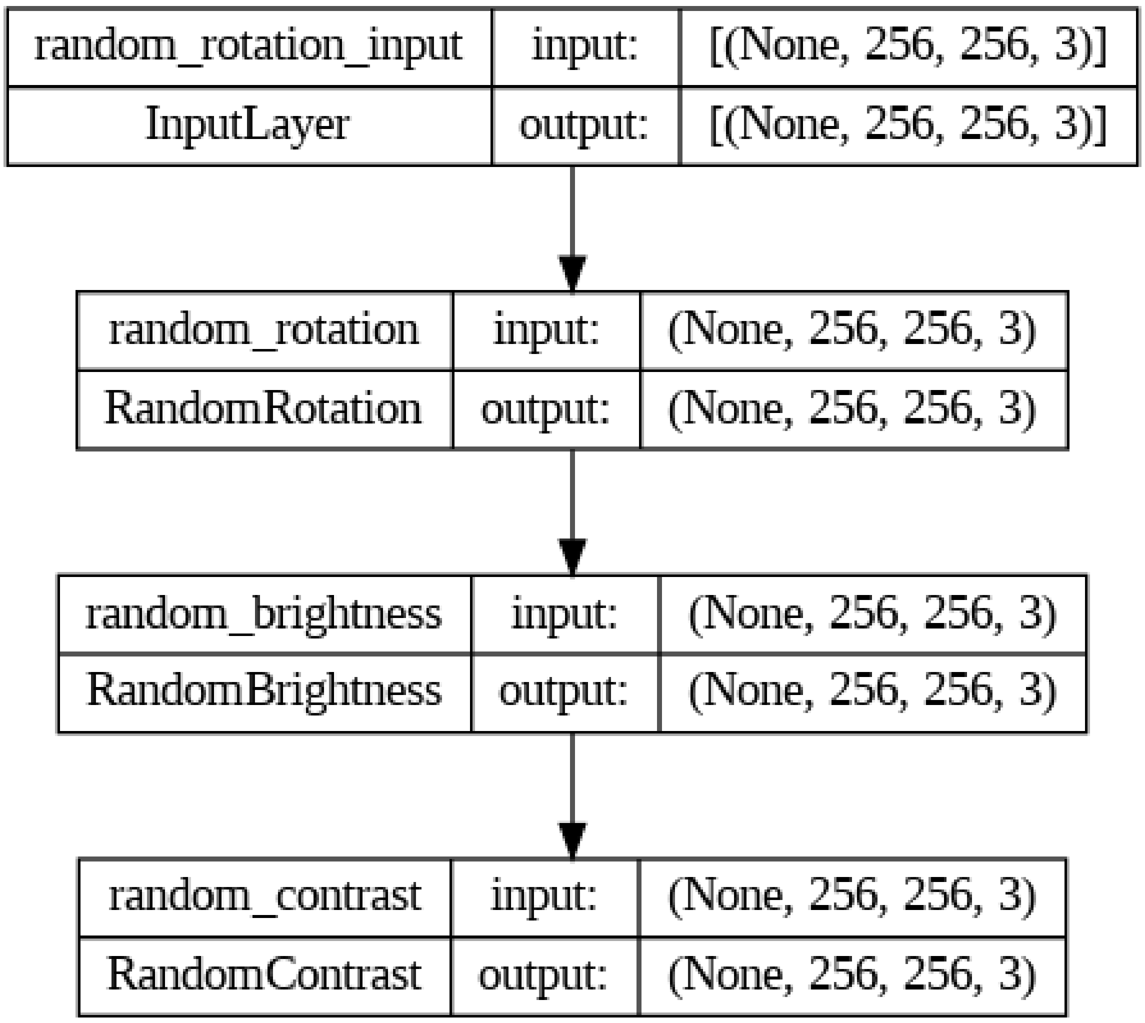
For this task, I am going to apply `RandomRotation`, `RandomContrast`, as well as `RandomBrightness` to our images.

In [ ]:
```
# Creating data augmentation pipeline
augmentation = tf.keras.Sequential(
    [
        tf.keras.layers.RandomRotation(
        factor = (-.25, .3),
        fill_mode = 'reflect',
        interpolation = 'bilinear',
        seed = seed),


        tf.keras.layers.RandomBrightness(
        factor = (-.45, .45),
        value_range = (0.0, 1.0),
        seed = seed),

        tf.keras.layers.RandomContrast(
        factor = (.5),
        seed = seed)
    ]
)
```

We can also use an `input_shape` as example to build the pipeline above and plot it below to illustrate how it looks.

In [ ]:
```
augmentation.build((None, 256, 256, 3)) # Building model
# Plotting model
tf.keras.utils.plot_model(augmentation,
                          show_shapes = True,
                          show_layer_names = True,
                          expand_nested = True)
```

Out[ ]:


random_rotation_input | input: | [(None, 256, 256, 3)]
InputLayer | output: | [(None, 256, 256, 3)]
random_rotation | input: | (None, 256, 256, 3)
RandomRotation | output: | (None, 256, 256, 3)
random_brightness | input: | (None, 256, 256, 3)
RandomBrightness | output: | (None, 256, 256, 3)
random_contrast | input: | (None, 256, 256, 3)
RandomContrast | output: | (None, 256, 256, 3)

We are going to attach this data augmentation pipeline to our convolutional neural network. It is important to remember that the data augmentation pipeline is inactive during testing, and the input samples will only be augmented during `fit()`, not when calling `predict()`.

Building the Convolutional Neural Network

To build the **Convolutional Neural Network** with Keras, we are going to use the Sequential class. This class allows us to build a linear stack of layers, which is essential for the creation of neural networks.

Besides the Convolutional, Pooling, and Fully-Connected Layers, which we have previously explored, I am also going to add the following layers to the network:

• **BatchNormalization**: This layer applies a transformation that maintains the mean output close to $0$ and the standard deviation close to $1$. It normalizes its inputs and is important to help convergence and generalization.

• **Dropout**: This layer randomly sets a fraction of input units to $0$ during training, which helps to prevent overfitting.

• **Flatten**: This layer transforms a multi-dimensional tensor into a one-dimensional tensor. It is used when transitioning from the Feature Learning segment — Convolutional and Pooling layers — to the fully-connected layers.

I plan to use different kernel sizes, both $3 \times 3$ and $5 \times 5$. This may allow the network to capture features at multiple scales.

I am also gradually increasing the dropout rates as we advance through the process and the increase in the number of kernels.

With that being said, let's go ahead and build our ConvNet.

In [ ]:

```
# Initiating model on GPU
with strategy.scope():
    model = Sequential()

    model.add(augmentation) # Adding data augmentation pipeline to th

    # Feature Learning Layers
    model.add(Conv2D(32,                  # Number of filters/Kernels
                     (3,3),               # Size of kernels (3x3 matr
                     strides = 1,         # Step size for sliding the
                     padding = 'same',    # 'Same' ensures that the o
                    input_shape = (256,256,3) # Input image shape
                    ))
    model.add(Activation('relu'))# Activation function
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.2))

    model.add(Conv2D(64, (5,5), padding = 'same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.2))

    model.add(Conv2D(128, (3,3), padding = 'same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.3))

    model.add(Conv2D(256, (5,5), padding = 'same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.3))

    model.add(Conv2D(512, (3,3), padding = 'same'))
    model.add(Activation('relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size = (2,2), padding = 'same'))
    model.add(Dropout(0.3))

    # Flattening tensors
    model.add(Flatten())

    # Fully-Connected Layers
    model.add(Dense(2048))
    model.add(Activation('relu'))
    model.add(Dropout(0.5))

    # Output Layer
    model.add(Dense(3, activation = 'softmax')) # Classification laye
```

• **optimizer**: In this parameter, we define the algorithms to adjust the weight updates. This is an important parameter, because choosing the right optimizer is essential to speed convergence. We are going to use `RMSprop`, which is the best optimizer I've found during the tests I ran.

• **loss**: This is the loss function we're trying to minimize during training. In this case, we are using `categorical_crossentropy`, which is a good choice for classification tasks with over two classes.

• **metrics**: This parameter defines the metric that will be used to evaluate performance during training and validation. Since our data is not heavily unbalanced, we may use `accuracy` for this, which is a very straightforward metric given by the following formula:

\begin{equation} \text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \end{equation}

In [ ]:

```
# Compiling model
model.compile(optimizer = tf.keras.optimizers.RMSprop(0.0001), # 1e-4
              loss = 'categorical_crossentropy', # Ideal for multicla
              metrics = ['accuracy']) # Evaluation metric
```

After compiling the model, I am going to define an **Early Stopping** and a **Model Checkpoint**.

Early Stopping serves the purpose of interrupting the training process when a certain metric stops improving over a period of time. In this case, I am going to configure the EarlyStopping method to monitor the accuracy in the test set, and stop the training process if we don't have any improvement on it after 5 epochs.

Model Checkpoint will ensure that only the best weights get saved, and we're also going to define the best weights according to the accuracy of the model in the test set.

In [ ]:

```
# Defining an Early Stopping and Model Checkpoints
early_stopping = EarlyStopping(monitor = 'val_accuracy',
                               patience = 5, mode = 'max',
                               restore_best_weights = True)

checkpoint = ModelCheckpoint('best_model.h5',
                             monitor = 'val_accuracy',
                             save_best_only = True)
```

We may now use `model.fit()` to start the training and testing process

```
In [ ]:
```

```
# Training and Testing Model
try:
    history = model.fit(
        train, epochs = 50,
        validation_data = test,
        callbacks = [early_stopping, checkpoint])
except Exception as e:
    print("An error occurred:", e)
```

```
Epoch 1/50
83/83 [==============================] - 84s 757ms/step - loss: 5.510
7 - accuracy: 0.5242 - val_loss: 9.4873 - val_accuracy: 0.3333
Epoch 2/50
83/83 [==============================] - 70s 745ms/step - loss: 2.626
6 - accuracy: 0.6044 - val_loss: 9.3485 - val_accuracy: 0.3333
Epoch 3/50
83/83 [==============================] - 71s 773ms/step - loss: 1.717
3 - accuracy: 0.6687 - val_loss: 6.1621 - val_accuracy: 0.3333
Epoch 4/50
83/83 [==============================] - 73s 818ms/step - loss: 1.170
0 - accuracy: 0.7156 - val_loss: 1.1241 - val_accuracy: 0.6133
Epoch 5/50
83/83 [==============================] - 70s 779ms/step - loss: 0.980
3 - accuracy: 0.7300 - val_loss: 3.9617 - val_accuracy: 0.5600
Epoch 6/50
83/83 [==============================] - 70s 787ms/step - loss: 0.739
1 - accuracy: 0.7753 - val_loss: 7.9041 - val_accuracy: 0.3733
Epoch 7/50
83/83 [==============================] - 64s 717ms/step - loss: 0.678
1 - accuracy: 0.8222 - val_loss: 2.7206 - val_accuracy: 0.5800
Epoch 8/50
83/83 [==============================] - 70s 761ms/step - loss: 0.601
3 - accuracy: 0.8313 - val_loss: 1.7578 - val_accuracy: 0.8000
Epoch 9/50
83/83 [==============================] - 71s 800ms/step - loss: 0.545
3 - accuracy: 0.8608 - val_loss: 0.9739 - val_accuracy: 0.9000
Epoch 10/50
83/83 [==============================] - 64s 717ms/step - loss: 0.528
7 - accuracy: 0.8623 - val_loss: 2.7072 - val_accuracy: 0.7267
Epoch 11/50
83/83 [==============================] - 67s 725ms/step - loss: 0.514
8 - accuracy: 0.8684 - val_loss: 2.2498 - val_accuracy: 0.7467
Epoch 12/50
```

```
83/83 [==============================] - 63s 702ms/step - loss: 0.425
9 - accuracy: 0.8805 - val_loss: 0.4981 - val_accuracy: 0.9000
Epoch 13/50
83/83 [==============================] - 69s 744ms/step - loss: 0.471
5 - accuracy: 0.8797 - val_loss: 0.8878 - val_accuracy: 0.9000
Epoch 14/50
83/83 [==============================] - 80s 913ms/step - loss: 0.418
8 - accuracy: 0.8949 - val_loss: 0.4669 - val_accuracy: 0.9533
Epoch 15/50
83/83 [==============================] - 66s 730ms/step - loss: 0.365
3 - accuracy: 0.9070 - val_loss: 1.2228 - val_accuracy: 0.8400
Epoch 16/50
83/83 [==============================] - 69s 752ms/step - loss: 0.388
9 - accuracy: 0.9070 - val_loss: 0.5287 - val_accuracy: 0.9200
Epoch 17/50
83/83 [==============================] - 66s 742ms/step - loss: 0.293
7 - accuracy: 0.9221 - val_loss: 0.5609 - val_accuracy: 0.9200
Epoch 18/50
83/83 [==============================] - 66s 739ms/step - loss: 0.306
7 - accuracy: 0.9221 - val_loss: 1.0246 - val_accuracy: 0.8733
Epoch 19/50
83/83 [==============================] - 73s 802ms/step - loss: 0.328
6 - accuracy: 0.9274 - val_loss: 0.4332 - val_accuracy: 0.9400
```

The highest accuracy for the testing set has been reached at the **19th epoch at 0.9200**, or **92%**, and didn't improve after that.

With the `history` object, we can plot two lineplots showing both the loss function and accuracy for both sets over epochs.

In [51]:

```python
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import numpy as np

# Dummy history data for example purposes
history = {
    'loss': np.random.rand(10),
    'val_loss': np.random.rand(10),
    'accuracy': np.random.rand(10),
    'val_accuracy': np.random.rand(10)
}

# Creating subplot
fig = make_subplots(rows=1, cols=2,
                    subplot_titles=['<b>Loss Over Epochs</b>', '<b>Ac
                    horizontal_spacing=0.2)

# Loss over epochs
train_loss = go.Scatter(x=list(range(len(history['loss']))),
                        y=history['loss'],
                        mode='lines',
                        line=dict(color='rgba(0, 67, 162, .75)', widt
                        name='Training',
                        showlegend=False)

val_loss = go.Scatter(x=list(range(len(history['val_loss']))),
                      y=history['val_loss'],
                      mode='lines',
                      line=dict(color='rgba(255, 132, 0, .75)', width
                      name='Test',
                      showlegend=False)

fig.add_trace(train_loss, row=1, col=1)
fig.add_trace(val_loss, row=1, col=1)

# Accuracy over epochs
train_acc = go.Scatter(x=list(range(len(history['accuracy']))),
                       y=history['accuracy'],
                       mode='lines',
                       line=dict(color='rgba(0, 67, 162, .75)', width
                       name='Training',
                       showlegend=True)

val_acc = go.Scatter(x=list(range(len(history['val_accuracy']))),
                     y=history['val_accuracy'],
                     mode='lines',
                     line=dict(color='rgba(255, 132, 0, .75)', width=
                     name='Test',
                     showlegend=True)

fig.add_trace(train_acc, row=1, col=2)
fig.add_trace(val_acc, row=1, col=2)

# Updating layout
fig.update_layout(
    title={'text': '<b>Loss and Accuracy Over Epochs</b>', 'x': 0.025
    margin=dict(t=100),
    height=500,
    width=1000,
    showlegend=True,
    plot_bgcolor='white',
    paper_bgcolor='white'
)

fig.update_yaxes(title_text='Loss', row=1, col=1)
fig.update_yaxes(title_text='Accuracy', row=1, col=2)

fig.update_xaxes(title_text='Epoch', row=1, col=1)
fig.update_xaxes(title_text='Epoch', row=1, col=2)

# Show figure using Plotly in Google Colab
import plotly.io as pio
pio.renderers.default = 'colab'  # Ensure the renderer is set to 'col

fig.show()
```

Leaf Disease Classification

Motivation

- As farmers and agriculture field are the important part of our life , farmers are the root level building blocks in the economy of any country . They work really heard for a whole season to grow a specific crop for survival of his family
- Sometimes these crops on which he dedicated his whole 3-6 months to nurture these crops got disease as result of which they can't sell their crops on the price he was expecting
- And He thinks if he knew these if he knew the plant disease before hand , he can use spefic pesticides and fertilizers to get over these disease
- What if we can use deep learning techniques to help famers to know about specific disease , so that they can be ready before harvestifying their crops

Description of Dataset

Here we announce the release of 4000 expertly curated images on healthy and infected leaves of crops plants through the existing online platform PlantVillage. We describe both the data and the platform. These data are the beginning of an ongoing, crowdsourcing effort to enable computer vision approaches to help solve the problem of yield losses in crop plants due to infectious diseases.

In [1]:

```
import os
from PIL import Image

# import data handling tools
import cv2
import numpy as np
import pandas as pd
import seaborn as sns
import itertools
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler , MinMaxScaler
from sklearn.pipeline import Pipeline
import warnings

with warnings.catch_warnings():
    warnings.simplefilter("ignore")

from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation, Flatten, Dropout, Dense
from keras import backend as K
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam
from keras.preprocessing import image
from keras.preprocessing.image import img_to_array
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

```
Using TensorFlow backend.
/opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/dt
ypes.py:516: FutureWarning: Passing (type, 1) or '1type' as a synonym
of type is deprecated; in a future version of numpy, it will be under
stood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/dt
ypes.py:517: FutureWarning: Passing (type, 1) or '1type' as a synonym
of type is deprecated; in a future version of numpy, it will be under
stood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/dt
ypes.py:518: FutureWarning: Passing (type, 1) or '1type' as a synonym
of type is deprecated; in a future version of numpy, it will be under
stood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/dt
ypes.py:519: FutureWarning: Passing (type, 1) or '1type' as a synonym
of type is deprecated; in a future version of numpy, it will be under
stood as (type, (1,)) / '(1,)type'.
Loading [MathJax]/extensions/Safe.js ype([("quint16", np.uint16, 1)])
```

```
/opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/dt
ypes.py:520: FutureWarning: Passing (type, 1) or '1type' as a synonym
of type is deprecated; in a future version of numpy, it will be under
stood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/opt/conda/lib/python3.6/site-packages/tensorflow/python/framework/dt
ypes.py:525: FutureWarning: Passing (type, 1) or '1type' as a synonym
of type is deprecated; in a future version of numpy, it will be under
stood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/opt/conda/lib/python3.6/site-packages/tensorboard/compat/tensorflow_
stub/dtypes.py:541: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of numpy, it will
be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/opt/conda/lib/python3.6/site-packages/tensorboard/compat/tensorflow_
stub/dtypes.py:542: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of numpy, it will
be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/opt/conda/lib/python3.6/site-packages/tensorboard/compat/tensorflow_
stub/dtypes.py:543: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of numpy, it will
be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/opt/conda/lib/python3.6/site-packages/tensorboard/compat/tensorflow_
stub/dtypes.py:544: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of numpy, it will
be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/opt/conda/lib/python3.6/site-packages/tensorboard/compat/tensorflow_
stub/dtypes.py:545: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of numpy, it will
be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/opt/conda/lib/python3.6/site-packages/tensorboard/compat/tensorflow_
stub/dtypes.py:550: FutureWarning: Passing (type, 1) or '1type' as a
synonym of type is deprecated; in a future version of numpy, it will
be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

Extracting the Data and Converting in DataFrame

In [ ]:

Loading [MathJax]/extensions/Safe.js

```
dataset_path = '/kaggle/input/plantdisease/PlantVillage'
selected_classes = ['Pepper__bell___Bacterial_spot', 'Potato___Late_b

data = []
labels = []
# Iterate through the dataset directory
for class_name in os.listdir(dataset_path):
    if class_name in selected_classes:
        class_dir = os.path.join(dataset_path, class_name)
        for img_name in os.listdir(class_dir):
            img_path = os.path.join(class_dir, img_name)
            data.append(img_path)
            labels.append(class_name)

df = pd.DataFrame({'data': data, 'label': labels})
```

Viewing df

In [ ]:

```
df
```

	data	lal
0	/kaggle/input/plantdisease/PlantVillage/Pepper...	Pepper__bell___Bacterial_sp
1	/kaggle/input/plantdisease/PlantVillage/Pepper...	Pepper__bell___Bacterial_sp
2	/kaggle/input/plantdisease/PlantVillage/Pepper...	Pepper__bell___Bacterial_sp
3	/kaggle/input/plantdisease/PlantVillage/Pepper...	Pepper__bell___Bacterial_sp
4	/kaggle/input/plantdisease/PlantVillage/Pepper...	Pepper__bell___Bacterial_sp
...	...	
3901	/kaggle/input/plantdisease/PlantVillage/Tomato...	Tomato_Late_blig
3902	/kaggle/input/plantdisease/PlantVillage/Tomato...	Tomato_Late_blig
3903	/kaggle/input/plantdisease/PlantVillage/Tomato...	Tomato_Late_blig
3904	/kaggle/input/plantdisease/PlantVillage/Tomato...	Tomato_Late_blig
3905	/kaggle/input/plantdisease/PlantVillage/Tomato...	Tomato_Late_blig

3906 rows × 2 columns

Analyzing the Data

In [ ]:

```
image = Image.open("/kaggle/input/plantdisease/PlantVillage/Pepper__b
width, height = image.size
print(f"Width: {width}, Height: {height}")
```

```
Width: 256, Height: 256
```

In [ ]:

```
plt.figure(figsize=(20, 15))

for i in range(5):
    plt.subplot(1, 5, i + 1)
    index = np.random.choice(df.index)
    filename = df.loc[index, 'data']
    category = df.loc[index, 'label']
    img = Image.open(filename)
    plt.imshow(img)
    plt.title(f'label: {category}')
    plt.axis('off')

plt.tight_layout()
plt.show()
```

label: Tomato_Late_blight
label: Tomato_Late_blight
label: Potato__Late_blight
label: Tomato_Late_blight
label: Tomato_Late_blight

Extracting HOG Features and Preparing Data

In [ ]:

```
def extract_hog_features(image):
    # Convert the image to grayscale using cv2
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    hog = cv2.HOGDescriptor()

    # Compute HOG features
    hog_features = hog.compute(gray_image)
```

Loading [MathJax]/extensions/Safe.js

```
    return hog_features.flatten()
```

Resizing the Data

In [ ]:

```
df_shuffled = df.sample(frac=1, random_state=42).reset_index(drop=Tru
batch_size = 32  # Adjust batch size based on memory constraints
features_list = []
labels_list = []

# Resize function to downsample images
def resize_image(image, new_size=(128, 128)):
    return cv2.resize(image, new_size)

for start in range(0, len(df_shuffled), batch_size):
    end = min(start + batch_size, len(df_shuffled))
    batch = df_shuffled[start:end]

    batch_features = []
    batch_labels = []

    for index, row in batch.iterrows():
        image = cv2.imread(row['data'])
        resized_image = resize_image(image)  # Resize image to smalle
        hog_features = extract_hog_features(resized_image)
        batch_features.append(hog_features)
        batch_labels.append(row['label'])

    features_list.extend(batch_features)
    labels_list.extend(batch_labels)
```

```
Shape of extracted HOG features: (3906, 34020)
```

In [ ]:

```
# Convert lists to NumPy arrays
features_array = np.array(features_list)
labels_array = np.array(labels_list)
label_encoder = LabelEncoder()
labels_encoded = label_encoder.fit_transform(labels_array)

print("Shape of extracted HOG features:", features_scaled.shape)
```

```
Shape of extracted HOG features: (3906, 34020)
```

In [ ]:

Loading [MathJax]/extensions/Safe.js

```
len(labels_encoded)
```

```
3906
```

In [ ]:

```
np.unique(labels_encoded)
```

```
array([0, 1, 2])
```

Test/Train Splitting

In [ ]:

```
X_train, X_test, y_train, y_test = train_test_split(features_array, l
```

In [ ]:

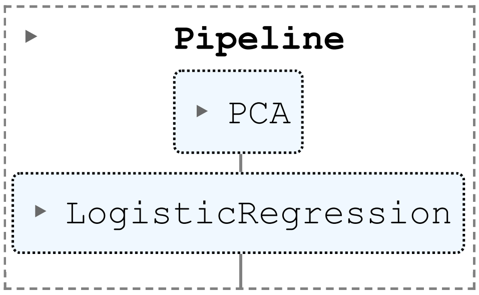
```
print(type(X_train), X_train.shape)
print(type(y_train), y_train.shape)
print(type(X_test), X_test.shape)
print(type(y_test), y_test.shape)
```

```
<class 'numpy.ndarray'> (2929, 34020)
<class 'numpy.ndarray'> (2929,)
<class 'numpy.ndarray'> (977, 34020)
<class 'numpy.ndarray'> (977,)
```

SVM

In [ ]:

```
lr_pipeline = Pipeline([
    ('pca', PCA(n_components=2100,random_state=42)),
    ('classifier', LogisticRegression(max_iter = 1000,random_state=42
])

lr_pipeline.fit(X_train, y_train)
```



Metrics

In [ ]:

```
predictions = lr_pipeline.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print(f"Accuracy: {accuracy:.4f}")
```

```
Accuracy: 0.8802
```

In [ ]:

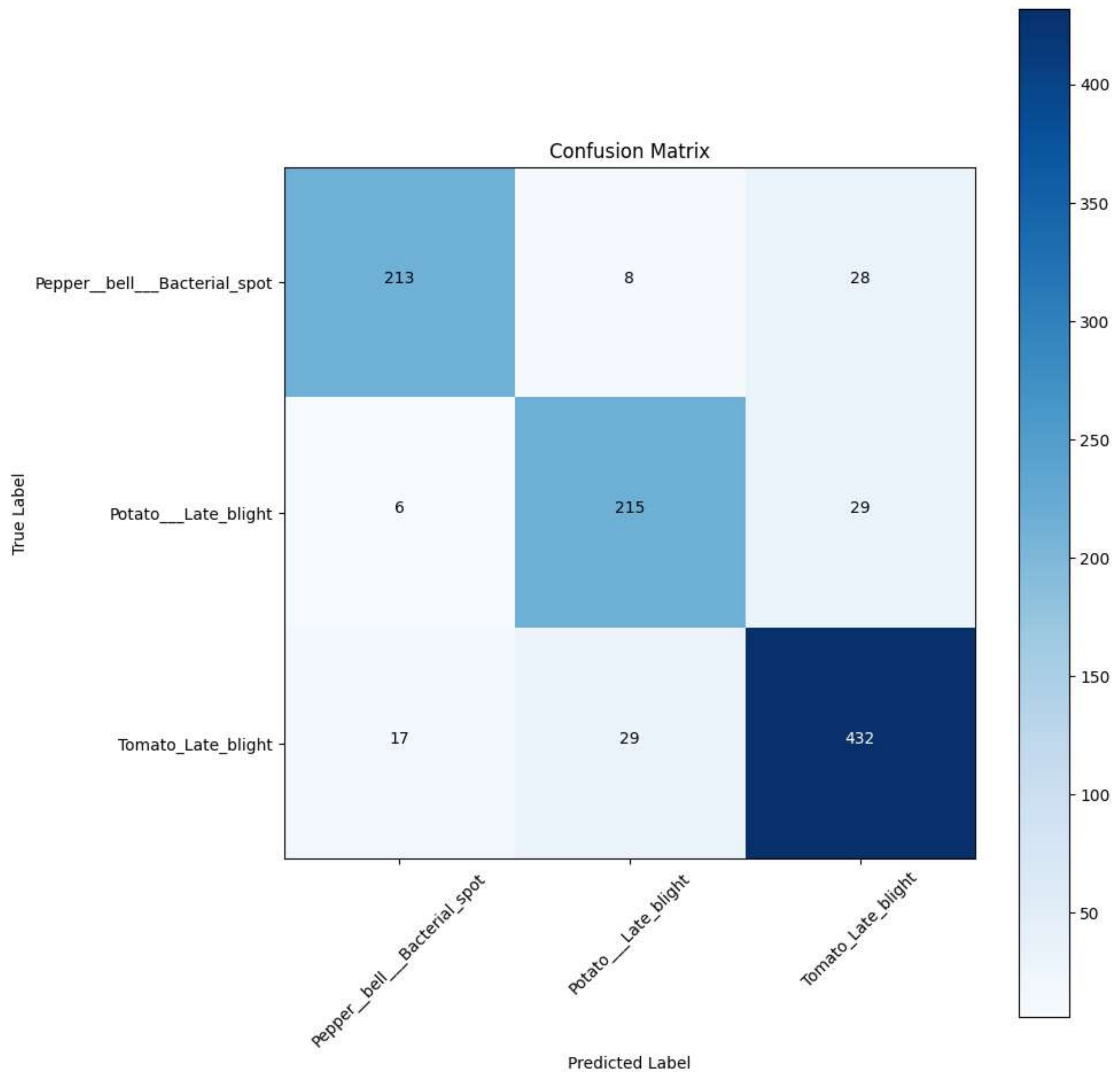
```
report = classification_report(y_test, predictions, output_dict=True,

# Convert the report to a pandas DataFrame for better visualization
report = pd.DataFrame(report).transpose()

print(report)
```

```
              precision    recall  f1-score     support
0              0.902542  0.855422  0.878351  249.000000
1              0.853175  0.860000  0.856574  250.000000
2              0.883436  0.903766  0.893485  478.000000
accuracy       0.880246  0.880246  0.880246    0.880246
macro avg      0.879718  0.873062  0.876136  977.000000
weighted avg   0.880562  0.880246  0.880183  977.000000
```

In [ ]:

```
classes = selected_classes



cm = confusion_matrix(y_test, predictions)

plt.figure(figsize= (10, 10))
plt.imshow(cm, interpolation= 'nearest', cmap= plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()

tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes, rotation= 45)
plt.yticks(tick_marks, classes)



thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1]))
    plt.text(j, i, cm[i, j], horizontalalignment= 'center', color= 'w

plt.tight_layout()
Loading [MathJax]/extensions/Safe.js l')
```

```
plt.xlabel('Predicted Label')


plt.show()
```

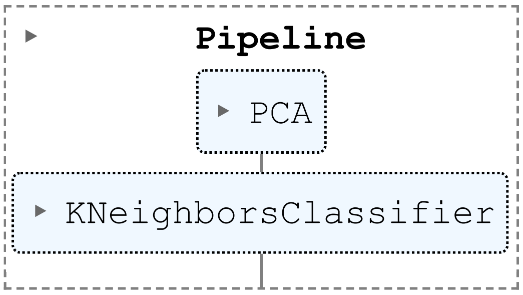


KNN

In [ ]:

```
knn_pipeline = Pipeline([
    ('pca', PCA(n_components=2100, random_state=42)),
    ('classifier', KNeighborsClassifier(n_neighbors=5))
])

knn_pipeline.fit(X_train, y_train)
```

Loading [MathJax]/extensions/Safe.js



In [ ]:

```
knn_predictions = knn_pipeline.predict(X_test)
knn_accuracy = accuracy_score(y_test, knn_predictions)
print(f"KNN Accuracy: {knn_accuracy:.4f}")

knn_report = classification_report(y_test, knn_predictions, output_di
knn_report_df = pd.DataFrame(knn_report).transpose()
print(knn_report_df)
```

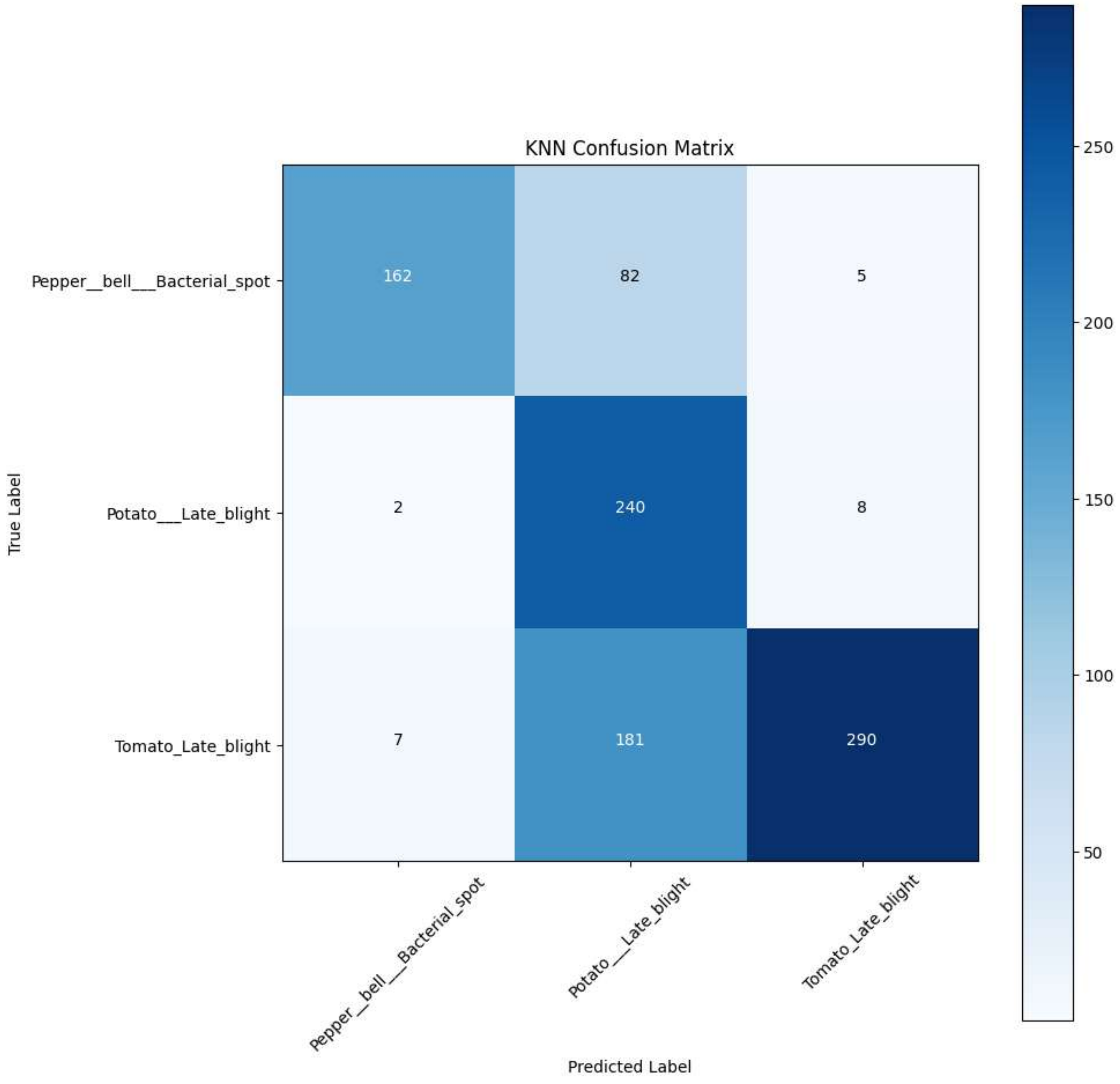
```
KNN Accuracy: 0.7083
              precision    recall  f1-score     support
0              0.947368  0.650602  0.771429  249.000000
1              0.477137  0.960000  0.637450  250.000000
2              0.957096  0.606695  0.742638  478.000000
accuracy       0.708291  0.708291  0.708291    0.708291
macro avg      0.793867  0.739099  0.717172  977.000000
weighted avg   0.831802  0.708291  0.723059  977.000000
```

In [ ]:

```
def plot_confusion_matrix(cm, classes, title='Confusion Matrix'):
    plt.figure(figsize=(10, 10))
    plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(title)
    plt.colorbar()

    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[
        plt.text(j, i, cm[i, j], horizontalalignment='center', color=

    plt.tight_layout()
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

                                         er.classes_
```

Loading [MathJax]/extensions/Safe.js

```
# Confusion Matrix for KNN
knn_cm = confusion_matrix(y_test, knn_predictions)
plot_confusion_matrix(knn_cm, classes, title='KNN Confusion Matrix')
```

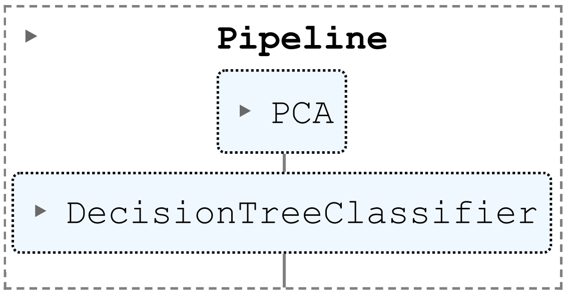


Decision Tree

In [ ]:

```
# Decision Tree Classifier
dt_pipeline = Pipeline([
    ('pca', PCA(n_components=2100, random_state=42)),
    ('classifier', DecisionTreeClassifier(random_state=42))
])

dt_pipeline.fit(X_train, y_train)
```

Loading [MathJax]/extensions/Safe.js



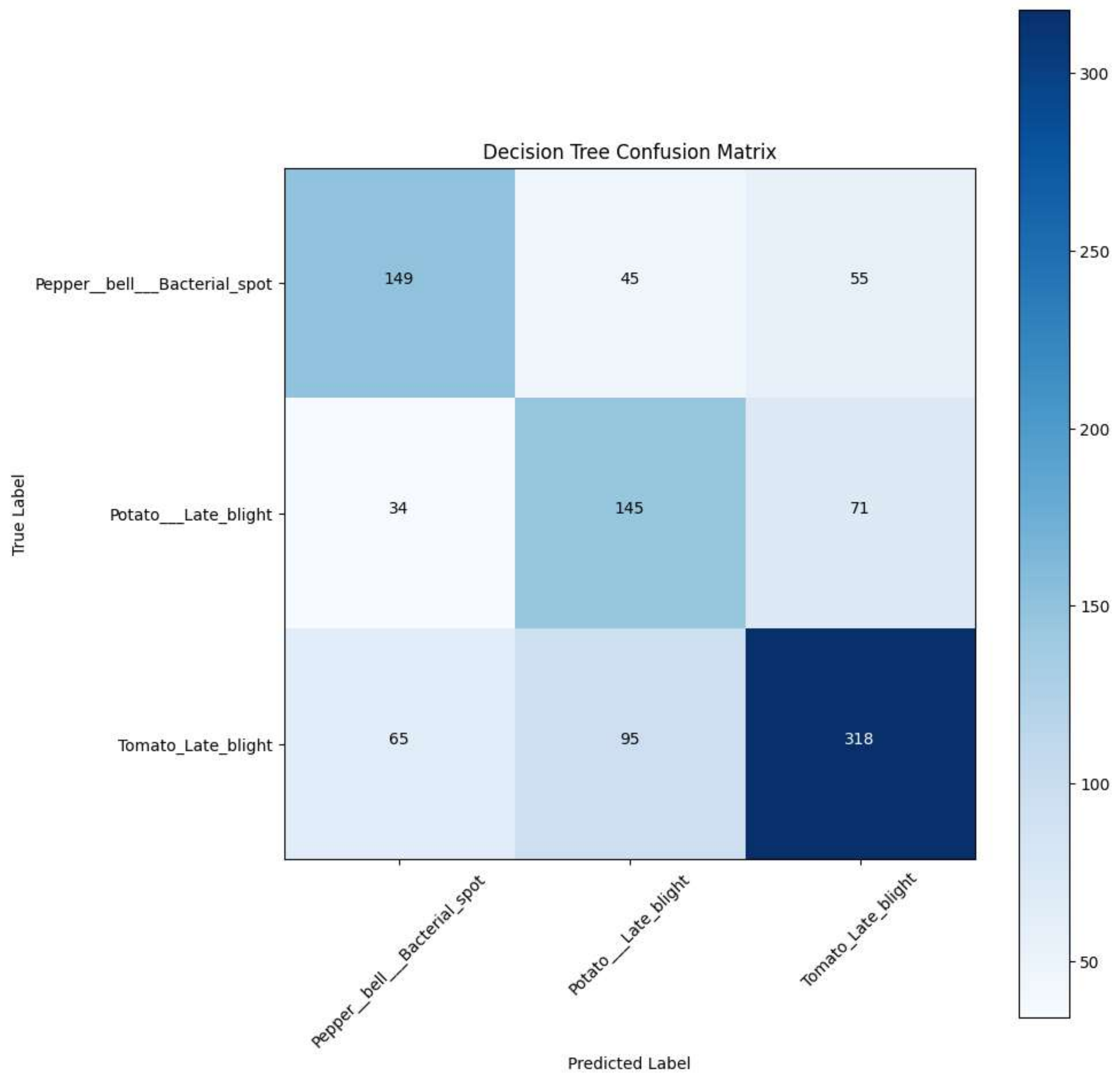
In [ ]:

```
dt_predictions = dt_pipeline.predict(X_test)
dt_accuracy = accuracy_score(y_test, dt_predictions)
print(f"Decision Tree Accuracy: {dt_accuracy:.4f}")

dt_report = classification_report(y_test, dt_predictions, output_dict
dt_report_df = pd.DataFrame(dt_report).transpose()
print(dt_report_df)
```

```
Decision Tree Accuracy: 0.6264
              precision    recall  f1-score     support
0              0.600806  0.598394  0.599598  249.000000
1              0.508772  0.580000  0.542056  250.000000
2              0.716216  0.665272  0.689805  478.000000
accuracy       0.626407  0.626407  0.626407    0.626407
macro avg      0.608598  0.614555  0.610486  977.000000
weighted avg   0.633721  0.626407  0.629008  977.000000
```

In [ ]:

```
# Confusion Matrix for Decision Tree
dt_cm = confusion_matrix(y_test, dt_predictions)
plot_confusion_matrix(dt_cm, classes, title='Decision Tree Confusion
```



CNN

In [2]:

```
EPOCHS = 25
INIT_LR = 1e-3
BS = 32
default_image_size = tuple((256, 256))
image_size = 0
directory_root = '/kaggle/input/plantdisease/PlantVillage'
width=256
height=256
depth=3
```

Function to convert images to array

Loading [MathJax]/extensions/Safe.js

In [3]:

```
def convert_image_to_array(image_dir):
    try:
        image = cv2.imread(image_dir)
        if image is not None :
            image = cv2.resize(image, default_image_size)
            return img_to_array(image)
        else :
            return np.array([])
    except Exception as e:
        print(f"Error : {e}")
        return None
```

Fetch images from directory

In [4]:

```
image_list, label_list = [], []
try:
    print("[INFO] Loading images ...")
    root_dir = listdir(directory_root)
    for directory in root_dir :
        # remove .DS_Store from list
        if directory == ".DS_Store" :
            root_dir.remove(directory)

    for plant_folder in root_dir :
        plant_disease_folder_list = listdir(f"{directory_root}/{plant_

        for disease_folder in plant_disease_folder_list :
            # remove .DS_Store from list
            if disease_folder == ".DS_Store" :
                plant_disease_folder_list.remove(disease_folder)

        for plant_disease_folder in plant_disease_folder_list:
            print(f"[INFO] Processing {plant_disease_folder} ...")
            plant_disease_image_list = listdir(f"{directory_root}/{pl

            for single_plant_disease_image in plant_disease_image_lis
                if single_plant_disease_image == ".DS_Store" :
                    plant_disease_image_list.remove(single_plant_dise

            for image in plant_disease_image_list[:200]:
                image_directory = f"{directory_root}/{plant_folder}/{
                if image_directory.endswith(".jpg") == True or image_
                    image_list.append(convert_image_to_array(image_di
                    label_list.append(plant_disease_folder)
```

Loading [MathJax]/extensions/Safe.js

```
    print("[INFO] Image loading completed")
except Exception as e:
    print(f"Error : {e}")
```

```
[INFO] Loading images ...
[INFO] Processing Tomato_Septoria_leaf_spot ...
[INFO] Processing Tomato_Late_blight ...
[INFO] Processing Potato___Late_blight ...
[INFO] Processing Tomato_healthy ...
[INFO] Processing Pepper__bell___healthy ...
[INFO] Processing Tomato_Spider_mites_Two_spotted_spider_mite ...
[INFO] Processing Tomato_Bacterial_spot ...
[INFO] Processing Tomato_Early_blight ...
[INFO] Processing Tomato__Tomato_YellowLeaf__Curl_Virus ...
[INFO] Processing Tomato__Tomato_mosaic_virus ...
[INFO] Processing Tomato__Target_Spot ...
[INFO] Processing Potato___Early_blight ...
[INFO] Processing Pepper__bell___Bacterial_spot ...
[INFO] Processing Tomato_Leaf_Mold ...
[INFO] Processing Potato___healthy ...
[INFO] Image loading completed
```

Get Size of Processed Image

In [5]:

```
image_size = len(image_list)
```

Transform Image Labels uisng Scikit LabelBinarizer

In [6]:

```
label_binarizer = LabelBinarizer()
image_labels = label_binarizer.fit_transform(label_list)
pickle.dump(label_binarizer,open('label_transform.pkl', 'wb'))
n_classes = len(label_binarizer.classes_)
```

Print the classes

In [7]:

```
print(label_binarizer.classes_)
```

```
['Pepper__bell___Bacterial_spot' 'Pepper__bell___healthy'
 'Potato___Early_blight' 'Potato___Late_blight' 'Potato___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_YellowLeaf__Curl_Virus' 'Tomato__Tomato_mosaic_viru
```

Loading [MathJax]/extensions/Safe.js

```
s'
 'Tomato_healthy']
```

In [8]:

```
np_image_list = np.array(image_list, dtype=np.float16) / 225.0
```

In [9]:

```
print("[INFO] Spliting data to train, test")
x_train, x_test, y_train, y_test = train_test_split(np_image_list, im
```

```
[INFO] Spliting data to train, test
```

In [10]:

```
aug = ImageDataGenerator(
    rotation_range=25, width_shift_range=0.1,
    height_shift_range=0.1, shear_range=0.2,
    zoom_range=0.2,horizontal_flip=True,
    fill_mode="nearest")
```

In [11]:

```
model = Sequential()
inputShape = (height, width, depth)
chanDim = -1
if K.image_data_format() == "channels_first":
    inputShape = (depth, height, width)
    chanDim = 1
model.add(Conv2D(32, (3, 3), padding="same",input_shape=inputShape))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(3, 3)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(Conv2D(64, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(Conv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(2, 2)))
Loading [MathJax]/extensions/Safe.js 5))
```

```
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(n_classes))
model.add(Activation("softmax"))
```

Model Summary

In [12]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 256, 256, 32)      896       
_________________________________________________________________
activation_1 (Activation)    (None, 256, 256, 32)      0         
_________________________________________________________________
batch_normalization_1 (Batch (None, 256, 256, 32)      128       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 85, 85, 32)        0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 85, 85, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 85, 85, 64)        18496     
_________________________________________________________________
activation_2 (Activation)    (None, 85, 85, 64)        0         
_________________________________________________________________
batch_normalization_2 (Batch (None, 85, 85, 64)        256       
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 85, 85, 64)        36928     
_________________________________________________________________
activation_3 (Activation)    (None, 85, 85, 64)        0         
_________________________________________________________________
batch_normalization_3 (Batch (None, 85, 85, 64)        256       
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 42, 42, 64)        0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 42, 42, 64)        0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 42, 42, 128)       73856     
_________________________________________________________________
activation_4 (Activation)    (None, 42, 42, 128)       0         
_________________________________________________________________
                       (Batch (None, 42, 42, 128)      512       
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 42, 42, 128)       147584    
_________________________________________________________________
activation_5 (Activation)    (None, 42, 42, 128)       0         
_________________________________________________________________
batch_normalization_5 (Batch (None, 42, 42, 128)       512       
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 21, 21, 128)       0         
_________________________________________________________________
dropout_3 (Dropout)          (None, 21, 21, 128)       0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 56448)             0         
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              57803776  
_________________________________________________________________
activation_6 (Activation)    (None, 1024)              0         
_________________________________________________________________
batch_normalization_6 (Batch (None, 1024)              4096      
_________________________________________________________________
dropout_4 (Dropout)          (None, 1024)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 15)                15375     
_________________________________________________________________
activation_7 (Activation)    (None, 15)                0         
=================================================================
Total params: 58,102,671
Trainable params: 58,099,791
Non-trainable params: 2,880
_________________________________________________________________
```

In [13]:

```
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
# distribution
model.compile(loss="binary_crossentropy", optimizer=opt,metrics=["acc
# train the network
print("[INFO] training network...")
```

```
[INFO] training network...
```

In [14]:

```
history = model.fit_generator(
    aug.flow(x_train, y_train, batch_size=BS),
    validation_data=(x_test, y_test),
    steps_per_epoch=len(x_train) // BS,
    epochs=EPOCHS, verbose=1
    )
```

Loading [MathJax]/extensions/Safe.js

```
73/73 [==============================] - 43s 583ms/step - loss: 0.201
2 - acc: 0.9365 - val_loss: 0.5047 - val_acc: 0.9022
Epoch 2/25
73/73 [==============================] - 35s 479ms/step - loss: 0.192
0 - acc: 0.9378 - val_loss: 0.8381 - val_acc: 0.8955
Epoch 3/25
73/73 [==============================] - 36s 491ms/step - loss: 0.161
7 - acc: 0.9469 - val_loss: 0.9227 - val_acc: 0.8950
Epoch 4/25
73/73 [==============================] - 36s 490ms/step - loss: 0.124
9 - acc: 0.9569 - val_loss: 0.9686 - val_acc: 0.8845
Epoch 5/25
73/73 [==============================] - 34s 470ms/step - loss: 0.105
2 - acc: 0.9611 - val_loss: 0.2461 - val_acc: 0.9453
Epoch 6/25
73/73 [==============================] - 36s 487ms/step - loss: 0.088
4 - acc: 0.9680 - val_loss: 0.1789 - val_acc: 0.9500
Epoch 7/25
73/73 [==============================] - 34s 468ms/step - loss: 0.079
8 - acc: 0.9707 - val_loss: 0.1902 - val_acc: 0.9558
Epoch 8/25
73/73 [==============================] - 35s 483ms/step - loss: 0.075
2 - acc: 0.9729 - val_loss: 0.5176 - val_acc: 0.9218
Epoch 9/25
73/73 [==============================] - 35s 476ms/step - loss: 0.084
8 - acc: 0.9698 - val_loss: 0.3263 - val_acc: 0.9277
Epoch 10/25
73/73 [==============================] - 35s 482ms/step - loss: 0.073
7 - acc: 0.9736 - val_loss: 0.2105 - val_acc: 0.9495
Epoch 11/25
73/73 [==============================] - 35s 481ms/step - loss: 0.069
9 - acc: 0.9746 - val_loss: 0.7867 - val_acc: 0.8999
Epoch 12/25
73/73 [==============================] - 35s 473ms/step - loss: 0.059
8 - acc: 0.9772 - val_loss: 0.1873 - val_acc: 0.9516
Epoch 13/25
73/73 [==============================] - 35s 478ms/step - loss: 0.060
4 - acc: 0.9791 - val_loss: 0.3801 - val_acc: 0.9381
Epoch 14/25
73/73 [==============================] - 34s 468ms/step - loss: 0.052
6 - acc: 0.9800 - val_loss: 0.5882 - val_acc: 0.9135
Epoch 15/25
73/73 [==============================] - 36s 487ms/step - loss: 0.051
2 - acc: 0.9810 - val_loss: 0.3619 - val_acc: 0.9357
Epoch 16/25
73/73 [==============================] - 35s 482ms/step - loss: 0.052
Loading [MathJax]/extensions/Safe.js _loss: 0.1112 - val_acc: 0.9642
```

```
Epoch 17/25
73/73 [==============================] - 34s 472ms/step - loss: 0.049
1 - acc: 0.9820 - val_loss: 0.6471 - val_acc: 0.9195
Epoch 18/25
73/73 [==============================] - 35s 484ms/step - loss: 0.047
8 - acc: 0.9824 - val_loss: 0.1848 - val_acc: 0.9582
Epoch 19/25
73/73 [==============================] - 34s 469ms/step - loss: 0.044
3 - acc: 0.9843 - val_loss: 0.1428 - val_acc: 0.9602
Epoch 20/25
73/73 [==============================] - 35s 483ms/step - loss: 0.045
5 - acc: 0.9832 - val_loss: 0.0754 - val_acc: 0.9755
Epoch 21/25
73/73 [==============================] - 35s 476ms/step - loss: 0.038
4 - acc: 0.9857 - val_loss: 0.5299 - val_acc: 0.9233
Epoch 22/25
73/73 [==============================] - 35s 481ms/step - loss: 0.046
3 - acc: 0.9842 - val_loss: 0.0925 - val_acc: 0.9701
Epoch 23/25
73/73 [==============================] - 35s 481ms/step - loss: 0.044
0 - acc: 0.9844 - val_loss: 1.2281 - val_acc: 0.8855
Epoch 24/25
73/73 [==============================] - 34s 472ms/step - loss: 0.039
6 - acc: 0.9847 - val_loss: 0.3955 - val_acc: 0.9337
Epoch 25/25
73/73 [==============================] - 35s 483ms/step - loss: 0.039
4 - acc: 0.9860 - val_loss: 0.1300 - val_acc: 0.9645
```

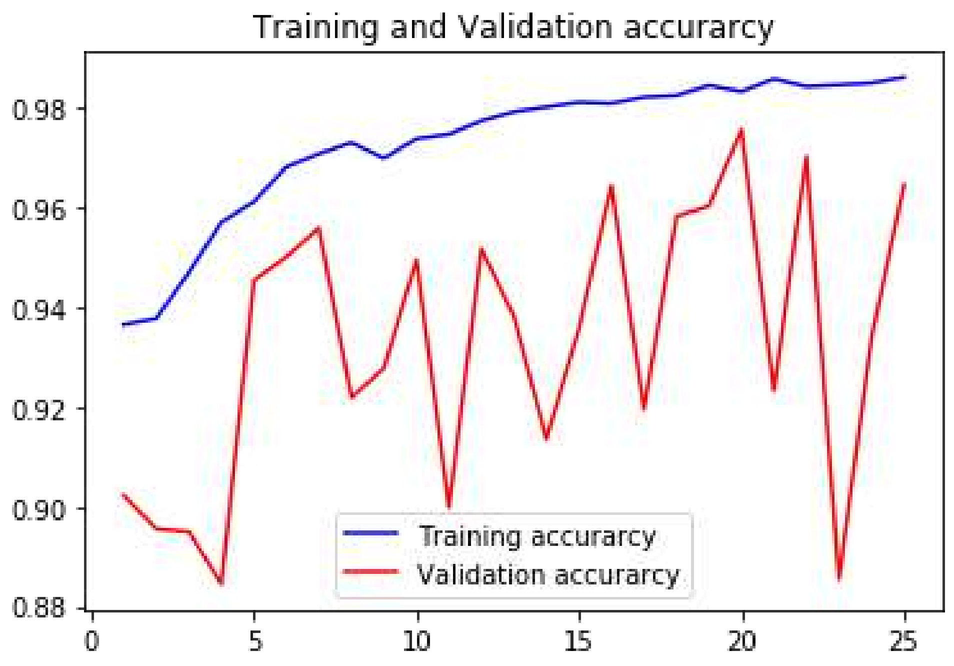
Plot the train and val curve

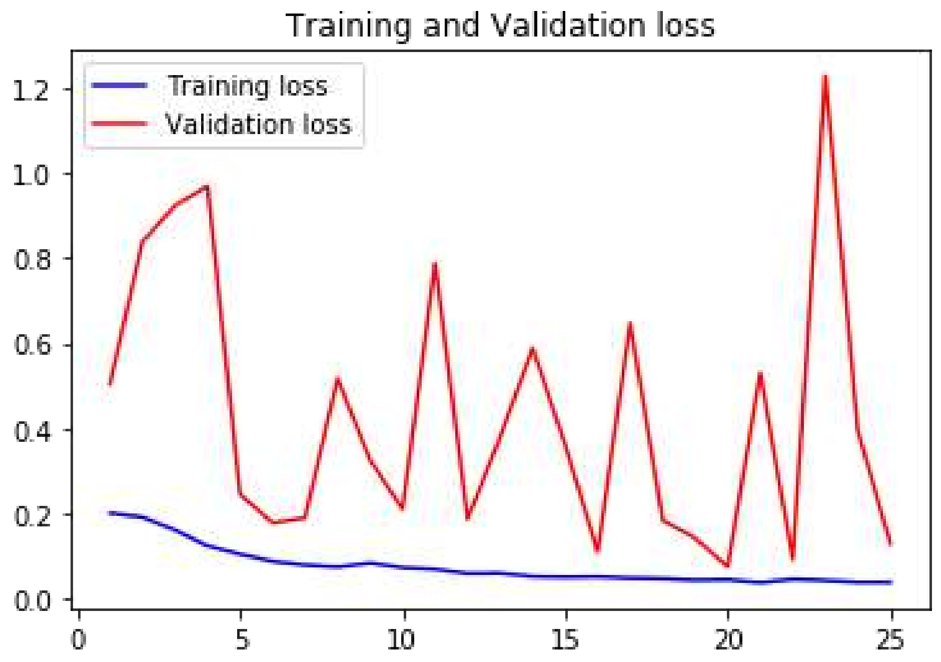
In [15]:

```python
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
#Train and validation accuracy
plt.plot(epochs, acc, 'b', label='Training accurarcy')
plt.plot(epochs, val_acc, 'r', label='Validation accurarcy')
plt.title('Training and Validation accurarcy')
plt.legend()

plt.figure()
#Train and validation loss
plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
```

Loading [MathJax]/extensions/Safe.js

```
plt.title('Training and Validation loss')
plt.legend()
plt.show()
```





Model Accuracy

In [16]:

```
print("[INFO] Calculating model accuracy")
scores = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {scores[1]*100}")
```

```
[INFO] Calculating model accuracy
591/591 [==============================] - 1s 2ms/step
Test Accuracy: 96.44670223221561
```

Loading [MathJax]/extensions/Safe.js

Project Analysis Report

Introduction

This report analyzes the performance of four different classifiers—Convolutional Neural Networks (CNN), Support Vector Machines (SVM), K-Nearest Neighbors (KNN), and Decision Trees (DT)—for plant disease recognition using image data. The analysis aims to identify which approach performs best under various scenarios and to understand the reasons behind their performance differences.

Datasets

- ***Plant Disease Recognition Dataset:*** Contains 1,530 images labeled as "Healthy," "Rust," and "Powdery."
- ***PlantVillage Dataset:*** Contains 4,000 images of healthy and infected leaves, focusing on the same three classes.

Classifiers and Methodology

The classifiers applied to primary datasets are:

CNN: A deep learning model particularly suited for image recognition tasks. I have write complete detail of CNN, starting from scratch. A beginner can easily understand by learning the notebook thoroughly. I have completely focused on CNN and make the notebook highly for CNN only. It comprises complete both theoretical and practical detail.

The classifiers applied to secondary datasets are:

1. CNN: A deep learning model particularly suited for image recognition tasks.

For the other 3 Models, I have take only 3 classes for training and testing to save memory and time. Because I've not enough resources to compile whole data set into np arrays and then evaluate the models.

2. SVM: A traditional machine learning algorithm effective in high-dimensional spaces.

3. KNN: An instance-based learning algorithm that classifies data points based on their neighbors.

4. DT: A non-parametric model that splits data into branches to make decisions.

Each classifier was tuned using various parameters, and their performance was evaluated using accuracy, precision, recall, F1-score, training time, and inference time. The results were visualized through accuracy and loss curves, confusion matrices, bar charts, and parameter sensitivity analysis.

Results and Discussion

1. Accuracy and Performance Metrics:

- CNN: Achieved the highest accuracy on both datasets. ***96% for both.*** It also showed the best performance in terms of precision, recall, and F1-score. The CNN was able to capture intricate patterns in the images, leading to superior classification performance.
- SVM: Performed well with ***88% Accuracy***, especially on the PlantVillage dataset, but not as well as the CNN. SVM showed good generalization but struggled with the complexity of image data compared to CNN.

- KNN: Had moderate performance with ***70% Accuracy***. It was effective for the Plant Disease Recognition Dataset. KNN's performance is highly dependent on the value of k and the distance metric used.
- DT: Provided decent results ***with 62% Accuracy*** but was prone to overfitting, especially on the smaller Plant Disease Recognition Dataset. Decision trees struggled with the complexity and variability in the image data.

2. Training and Inference Time:

- ***CNN:*** Required significant training time (about one hour, collectively for both datasets) but was efficient during inference. The training process involves numerous epochs (total 50 but stopped at 19, as performance become repetitive) and a large amount of data processing, but once trained, the CNN can quickly classify new images.
- ***SVM:*** Had relatively faster training times compared to CNN but was slower during inference, particularly with larger datasets due to the need to compute distances to support vectors.
- ***KNN:*** Very fast training time since it is a lazy learner, but inference was slow, especially with larger datasets, as it requires computing the distance to all training samples.
- ***DT:*** Training time was fast, but the inference time was highly variable depending on the tree depth and the dataset size.

Which Approach is Better and Why?

1. CNN:

- Best for Complex Image Data: CNNs excel in scenarios involving complex image data with high variability and intricate patterns. They are the best choice for tasks requiring high accuracy and detailed feature extraction.
- Deep Learning Advantages: The ability of CNNs to learn hierarchical representations makes them superior for image classification tasks.
- Scalability: CNNs can be scaled with more data and deeper architectures to improve performance further.

2. SVM:

- Effective for Smaller Datasets: SVMs are suitable for scenarios with smaller datasets where training a deep learning model might not be feasible.
- Good Generalization: SVMs provide good generalization and can perform well with high-dimensional data.
- Less Computationally Intensive: SVMs require less computational power compared to training deep learning models, making them practical for environments with limited resources.

3. KNN:

- Simple and Intuitive: KNN is effective for smaller, simpler datasets and can be a good baseline model.
- Parameter Sensitivity: KNN's performance heavily depends on the choice of k and the distance metric, making it less stable for larger and more complex datasets.
- Memory Intensive: KNN requires storing the entire training set, which can be impractical for large datasets.

4. DT:

- Interpretable and Fast: Decision Trees are highly interpretable and have fast training times, making them useful for exploratory data analysis and scenarios where model interpretability is crucial.
- Overfitting Prone: They tend to overfit, especially with small datasets or high-dimensional data, unless pruned or regularized.

Conclusion

In conclusion, CNNs are the best choice for plant disease recognition tasks involving complex image data due to their superior ability to learn and generalize from intricate patterns. SVMs offer a good alternative for smaller datasets or environments with limited computational resources. KNN and DT can be useful for quick, simple analyses or as baseline models, but they lack the robustness and scalability of CNNs. The choice of classifier should be based on the specific requirements of the task, including the size and complexity of the dataset, the need for interpretability, and available computational resources.