PRJ381: FINAL REPORT

Kenneth Mc Ferrier 577503

Leon Botha 576378

Kealeboga Molefe 577482

Keneilwe Mpho Ramatsui 577669

Kaylee Carstens 577331

Table of Contents

Introduction

This report describes the creation of a machine learning-based system that categorises Aloe Ferox images into four phenological stages: flower, bud, fruit, and no evidence. The objective was to develop an effective and scalable method for automating phenological analysis for researchers and agricultural experts.

Problem statement

Phenology studies are essential for understanding plant lifecycle stages and how they interact with environmental changes. Manually annotating phenological stages takes effort and is prone to inaccuracy. The lack of automated tools for large-scale phenological classification is a considerable challenge to researchers.

What we did

Me and my team created a machine learning model that can classify which stage of the Aloe Pherox's life cycle it is. We did this by developing a pipeline for data collection, preprocessing, model training and deployment. To make our model usable we created a user friendly interface.

Workflow and implementation

Image collection:

First we extracted the provided images from the excel sheet and automatically stored those images into their respective categories.

Tools used:

- **Python:** for data extraction and file management
- **Pandas:** to read the URLs from the excel sheet.
- **Requests:** to download said images.
- **TQDM:** for progress tracking during downloads.

Image splitting:

We organized all of the images into train, validation and test directories. In each of these directories is the different classification categories aswel.

The images were split as shown here:

- Training set: 70%
- Validation set: 15%
- Testing set: 15%

Tools used:

- **OS Module:** to handle file system operations.
- **Shutil:** to move the categories into the appropriate directories.
- **Python random library:** for shuffling and splitting images.

Image preprocessing:

In this step we normalized and augmented the images to improve model generalization. We used augmentation techniques like flipping, rotation, zoom, brightness and contrast adjustments.

Tools used:

- **Tensorflow/Keras:** to create datasets and apply preprocessing.

- **Data Augmentation layers:** for flips, rotations, zoom, brightness and contrast adjustments.
- **Rescaling layer:** to normalize pixel values for better model performance.

Model training:

We then went on to create our machine learning model. Our model consist of a base model called EfficientNetB0 that we customized, added additional layers onto and fine tuned.

Tools used:

- **EfficientNetB0:** pre-trained model leveraged for transfer learning.
- **AdamW Optimizer:** for efficient gradient based optimization.
- **EarlyStopping callback:** this stops training when no further improvement during training is observed.
-

User Interface development:

We created our custom UI using Unity. This allows users to upload their own images to use our model. We used Unity because this will make it a lot easier for when we want to launch our program on android devices.

Tools used:

- **Unity:** To create our UI.

Model deployment and UI integration:

We created a REST API using Fast to serve predictions that our model calculated. With our API script we were able to integrate our model to our UI very seamlessly.

Tools used:

- **Fast:** for API development.
- **Tensorflow:** for model serving.
- **PIL:** for image processing.

Final results and outcomes

Final results:

Our model got an overall accuracy of 56% but accuracy will vary with different images. This program can also be used on unseen images.

Outcome:

We developed an intuitive UI that allows users to upload images of the Aloe Ferox and classifies the image into its correct category.

Evolution of our model

Initial scripts and Initial results:

Download and organizing script:

The main goal of this script is to download and organize the images into the appropriate folders. **(This script stayed the same throughout our model building process.)**

The libraries we used for this script:

- **pandas:** This was used to read the data from the excel file.
- **requests:** Used to download the images from the URLs.
- **os:** To manage directories and file paths.
- **tqdm:** To show us a progress bar during the image downloading process.

download_and_organize.py > ...

```python
import pandas as pd
import os
import requests
from tqdm import tqdm

file_path = 'C:\\Users\\user-pc\\Documents\\PRJ381(2)\\data\\Aloe ferox PHENOLOGY.xlsx'
base_dir = 'Images'
classes = ['flower', 'bud', 'fruit', 'no_evidence']

for split in ['train', 'validation', 'test']:
    for cls in classes:
        os.makedirs(os.path.join(base_dir, split, cls), exist_ok=True)

def download_images(sheet_name, class_name):

    df = pd.read_excel(file_path, sheet_name=sheet_name)

    if 'image_url' not in df.columns:
        print(f"Sheet '{sheet_name}' does not contain an 'image_url' column.")
        return

    for index, row in tqdm(df.iterrows(), total=df.shape[0], desc=f"Downloading {class_name} images"):
        image_url = row['image_url']
        image_name = f"{class_name}_{index}.jpg"
        split = 'train'

        image_path = os.path.join(base_dir, split, class_name, image_name)

        try:
            response = requests.get(image_url, stream=True, timeout=10)
            if response.status_code == 200:
                with open(image_path, 'wb') as file:
                    for chunk in response.iter_content(1024):
                        file.write(chunk)
            else:
                print(f"Failed to download {image_url}: Status code {response.status_code}")
        except Exception as e:
            print(f"Failed to download {image_url}: {e}")

download_images('FLOWERS', 'flower')
download_images('BUDS', 'bud')
download_images('FRUIT', 'fruit')
download_images('No Evidence', 'no_evidence')
```

Steps in this script:

1. Define paths:
    - Specifies the path to the excel file that was provided to us.
    - It defines the “base_dir” (Images) and classes (like “flower” and “bud”) to organize the images into different categories.

2. Creation of base directories:
    - We create folders for training, validation and testing splits for each class.

3. Downloading the images:
    - The excel file gets read through to check for the “image_url” column and then downloads the images to the appropriate folders.
    - If any of the images fails to download, it just prints an error message and then continues downloading the rest of the images.

Split data script:

In this script the images we downloaded gets split into training, validation and test sets.

(This script stayed the same throughout our model building process.)

The libraries we used for this script:

- **os and shutil:** For the file and directory operations.
- **random:** To shuffle the images for splitting.

split_data.py > ...

```python
import os
import shutil
import random

base_dir = 'Images'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

os.makedirs(validation_dir, exist_ok=True)
os.makedirs(test_dir, exist_ok=True)

val_split = 0.15
test_split = 0.15

for class_name in os.listdir(train_dir):
    class_path = os.path.join(train_dir, class_name)

    if os.path.isdir(class_path):
        os.makedirs(os.path.join(validation_dir, class_name), exist_ok=True)
        os.makedirs(os.path.join(test_dir, class_name), exist_ok=True)

        images = os.listdir(class_path)
        random.shuffle(images)

        total_images = len(images)
        val_size = int(total_images * val_split)
        test_size = int(total_images * test_split)

        val_images = images[:val_size]
        test_images = images[val_size:val_size + test_size]

        for image in val_images:
            src_path = os.path.join(class_path, image)
            dst_path = os.path.join(validation_dir, class_name, image)
            shutil.move(src_path, dst_path)

        for image in test_images:
            src_path = os.path.join(class_path, image)
            dst_path = os.path.join(test_dir, class_name, image)
            shutil.move(src_path, dst_path)

print("Images have been split into train, validation, and test sets.")
```

Steps in this script:

1. Defining paths:
    - Defines the paths for the training, validation and testing directories.

2. Creation of the directories:
    - This ensures that the directories for the validation and test sets exist.

3. Proportioning the validation and test sets:
    - This specifies that 15% of the images is for validation (val_split) and 15% is for testing (test_split).

4. Splitting the images:

- The images get shuffled in each class folder to ensure that the images are in a random order.
- The images then get split into validation and test sets and gets moved to their respective folders using shutil.move().

What the different sets get used for:

- Train set: Used for the model training.
- Validation set: This is used to evaluate the model while training to prevent overfitting.
- Test set: This is used for final evaluation to see how well the model generalizes to new , unseen data.

Initial Preprocessing script:

This script loads the images, applies data augmentation, normalizes them, and saves the preprocessed images.

The libraries we used for this script:

- **tensorflow:** To handle data loading, augmentation and processing.
- **os:** To handle the file paths.

PreProcessing.py > ...

```python
import tensorflow as tf
import os

base_dir = 'Images'
train_dir = f'{base_dir}/train'
validation_dir = f'{base_dir}/validation'
test_dir = f'{base_dir}/test'

preprocessed_base_dir = 'Preprocessed_Images'
train_save_dir = f'{preprocessed_base_dir}/train'
validation_save_dir = f'{preprocessed_base_dir}/validation'
test_save_dir = f'{preprocessed_base_dir}/test'

os.makedirs(train_save_dir, exist_ok=True)
os.makedirs(validation_save_dir, exist_ok=True)
os.makedirs(test_save_dir, exist_ok=True)

train_dataset = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

validation_dataset = tf.keras.utils.image_dataset_from_directory(
    validation_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

test_dataset = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

normalization_layer = tf.keras.layers.Rescaling(1.0 / 255)

data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip('horizontal'),
    tf.keras.layers.RandomRotation(0.3),
    tf.keras.layers.RandomZoom(0.3),
    tf.keras.layers.RandomBrightness(0.3),
    tf.keras.layers.RandomContrast(0.3)
])

def save_preprocessed_images(dataset, save_dir, augment=False):
    for batch_index, (images, labels) in enumerate(dataset):
        augmented_images = data_augmentation(images) if augment else images
        normalized_images = normalization_layer(augmented_images)

        for i in range(len(normalized_images)):
            label = tf.argmax(labels[i]).numpy()
            label_dir = os.path.join(save_dir, str(label))
            os.makedirs(label_dir, exist_ok=True)
            file_path = os.path.join(label_dir, f'image_{batch_index * len(normalized_images) + i}.jpg')
            try:
                tf.keras.preprocessing.image.save_img(file_path, normalized_images[i])
            except Exception as e:
                print(f"Error saving image {file_path}: {e}")

save_preprocessed_images(train_dataset, train_save_dir, augment=True)

save_preprocessed_images(validation_dataset, validation_save_dir, augment=False)
save_preprocessed_images(test_dataset, test_save_dir, augment=False)

print(f"Preprocessed images saved in '{preprocessed_base_dir}' directory.")
```

Steps in this script:

1. Defining the paths:
    - Defines the paths for the train, validation and test directories.
    - New directories for saving the preprocessed images gets created (Preprocessed_Images).

2. Loading the datasets:
    - The script uses '**image_dataset_from_directory()**' to load the images from the train, validation and test directories.
    - All the images are resized to (224, 224) pixels.
    - The script loads the images in batches of 32 to efficiently train the model.

3. Normalization Layer:
    - A rescaling layer gets applied to normalize the pixel values from [0, 255] to [0, 1] by dividing each value by 255.

4. Data gets Augmented:
    - The following augmentation techniques are used to artificially expand the dataset by introducing variations and helps the model generalize better:
        - Random Flip
        - Random Rotation
        - Random Zoom
        - Random Brightness
        - Random Contrast

5. Save preprocessed images:
    - The augmented images are saved in the "train" folder and the non-augmented images are saved in the "validation" and "test" folders.
    - The script iterates through the dataset and saves each preprocessed image using '**tf.keras.preprocessing.image.save_img()**'.

Initial Model training script:

This script is responsible for creating the actual machine learning model using the preprocessed images. This script also evaluates the performance of the model. Transfer learning was used to leverage pre-trained features from EfficientNet, thereby speeding up and improving the model's accuracy.

The libraries we used for this script:

- **tensorflow, keras, EfficientNetB0, ImageDataGenerator:** For building, training and evaluating the model.

ModelTraining.py > ...

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

base_dir = 'Preprocessed_Images'
train_dir = f'{base_dir}/train'
validation_dir = f'{base_dir}/validation'
test_dir = f'{base_dir}/test'

batch_size = 32

datagen = ImageDataGenerator(rescale=1./255,
                             rotation_range=40,
                             width_shift_range=0.2,
                             height_shift_range=0.2,
                             shear_range=0.2,
                             zoom_range=0.2,
                             horizontal_flip=True,
                             fill_mode='nearest')

train_datagen = datagen.flow_from_directory(train_dir, target_size=(224, 224), batch_size=batch_size, class_mode='categorical')
validation_datagen = datagen.flow_from_directory(validation_dir, target_size=(224, 224), batch_size=batch_size, class_mode='categorical')
test_datagen = datagen.flow_from_directory(test_dir, target_size=(224, 224), batch_size=batch_size, class_mode='categorical')

base_model = EfficientNetB0(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

base_model.trainable = False

x = base_model.output
x = tf.keras.layers.GlobalAveragePooling2D()(x)
x = tf.keras.layers.Dense(128, activation='relu', kernel_regularizer=keras.regularizers.l2(0.001))(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.Dropout(0.5)(x)
x = tf.keras.layers.Dense(4, activation='softmax')(x)

model = tf.keras.Model(inputs=base_model.input, outputs=x)

model.compile(optimizer=keras.optimizers.AdamW(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=20,
    restore_best_weights=True
)

history = model.fit(train_datagen, epochs=30, validation_data=validation_datagen, verbose=2, callbacks=[early_stopping])

base_model.trainable = True
for layer in base_model.layers[:len(base_model.layers) // 2]:
    layer.trainable = False

model.compile(optimizer=keras.optimizers.AdamW(learning_rate=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_fine = model.fit(train_datagen, epochs=20, validation_data=validation_datagen, verbose=2, callbacks=[early_stopping])

loss, accuracy = model.evaluate(test_datagen)
print(f"Test Accuracy: {accuracy:.2f}")

model.save('plant_lifecycle_classifier.keras')
```

Steps in this script:

1. The suppression of the warnings:
    - During the installation of Tensorflow and Python something must have went wrong because we keep getting warnings when importing them.
    - We wrote code to suppress the warnings to avoid excessive console logs.

2. Defining the paths:
    - This script uses the preprocessed images from the "Preprocessed_Images" folder.

3. Data generators:
    - The script uses ImageDataGenerator to generate batches of image data with real time data augmentation.
    - Random transformations like rotation, width/height shift, shear, zoom, etc. are used.

4. Loading EfficientNetB0:
    - A pre-trained EfficientNetB0 model which has been trained on a large dataset (ImageNet) gets loaded. This model helps to quickly build an effective model sine it has already learned useful features.
    - The "Include Top = False" part removes the classification layer, so we can add a custom head for our four classes.

5. Adding custom layers:
    - We add a GlobalAveragePooling layer followed by several dense layers with dropout and batch normalization.
    - The final layer has four output nodes with softmax activation to output probabilities for each of the four classes.

6. Compile the model:
    - The script uses the AdamW optimizer with a learning rate of 1e-4.
    - The loss function is categorical crossentropy, which is suitable for multi-class classification problems.

7. Early stopping callback:
    - This callback stops training if the validation loss stops improving fro 20 epochs, preventing overfitting.

8. Training the model:
    - The model gets trained for a total of 30 epochs using "train_datagen" and evaluates it using "validation_datagen".

9. Fine-tuning the model:
    - This unfreezes the base model layers, making them trainable.
    - The model gets compiles with a lower learning rate (1e-5) to adjust the features learned by the EfficientNet base.
    - The model gets fine-tuned for an additional 20 epochs.

10. Evaluating the model:
    - The model gets evaluated on the test dataset and the accuracy gets printed.

11. Saving the model:
    - The model gets saved to the "plant_lifecycle_classifier.keras" file.

Initial results:

Initially the model the model started with a low accuracy of 23% for training. After a lot of epochs , the training accuracy did start to increase, but the validation accuracy remained stuck at around 31%. The official accuracy at the end of the initial run was 25%.

The issue we faced could have been because the model was underfitting, leading to the model not being able to learn enough from the data.

Possible reasons for underfitting:

- The dataset size might be too small for a complex model like EfficientNet, which requires a lot of data.
- The data augmentation might have been too aggressive.

```
Epoch 10/20
56/56 - 45s - 809ms/step - accuracy: 0.2649 - loss: 1.9316 - val_accuracy: 0.3079 - val_loss: 1.8539
Epoch 11/20
56/56 - 46s - 814ms/step - accuracy: 0.2598 - loss: 1.9711 - val_accuracy: 0.2507 - val_loss: 1.7542
Epoch 12/20
56/56 - 45s - 812ms/step - accuracy: 0.2818 - loss: 1.8791 - val_accuracy: 0.2779 - val_loss: 1.8282
Epoch 13/20
56/56 - 45s - 811ms/step - accuracy: 0.2750 - loss: 1.9266 - val_accuracy: 0.2534 - val_loss: 1.7920
Epoch 14/20
56/56 - 46s - 813ms/step - accuracy: 0.2649 - loss: 1.9139 - val_accuracy: 0.2316 - val_loss: 1.8385
Epoch 15/20
56/56 - 47s - 835ms/step - accuracy: 0.2626 - loss: 1.9142 - val_accuracy: 0.2507 - val_loss: 1.6994
Epoch 16/20
56/56 - 46s - 815ms/step - accuracy: 0.2716 - loss: 1.9223 - val_accuracy: 0.2316 - val_loss: 1.7776
Epoch 17/20
56/56 - 45s - 808ms/step - accuracy: 0.2699 - loss: 1.9039 - val_accuracy: 0.2180 - val_loss: 1.9389
Epoch 18/20
56/56 - 45s - 810ms/step - accuracy: 0.2626 - loss: 1.9309 - val_accuracy: 0.2316 - val_loss: 1.8255
Epoch 19/20
56/56 - 46s - 814ms/step - accuracy: 0.2683 - loss: 1.8753 - val_accuracy: 0.2289 - val_loss: 1.9818
Epoch 20/20
56/56 - 45s - 812ms/step - accuracy: 0.2507 - loss: 1.9180 - val_accuracy: 0.2289 - val_loss: 1.8102
12/12 ━━━━━━━━━━━━━━━━━━━━ 6s 474ms/step - accuracy: 0.2750 - loss: 1.7581
Test Accuracy: 0.25
```

Second iteration:

2nd Iteration of the Model training script:

ModelTraining.py > ...

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications import EfficientNetB0
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import warnings
warnings.filterwarnings("ignore")

base_dir = 'Images'
train_dir = f'{base_dir}/train'
validation_dir = f'{base_dir}/validation'
test_dir = f'{base_dir}/test'

batch_size = 16

train_dataset = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
).repeat()

validation_dataset = tf.keras.utils.image_dataset_from_directory(
    validation_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
)

test_dataset = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
)

AUTOTUNE = tf.data.AUTOTUNE
train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)
```

```python
num_train_samples = 1769
num_validation_samples = 376
steps_per_epoch = num_train_samples // batch_size
validation_steps = num_validation_samples // batch_size

data_augmentation = keras.Sequential([
    keras.layers.RandomFlip("horizontal"),
    keras.layers.RandomRotation(0.2),
    keras.layers.RandomZoom(0.2)
])
train_dataset = train_dataset.map(lambda x, y: (data_augmentation(x, training=True), y), num_parallel_calls=AUTOTUNE)

base_model = EfficientNetB0(input_shape=(224, 224, 3), include_top=False, weights='imagenet')
base_model.trainable = False

model = keras.Sequential([
    base_model,
    keras.layers.GlobalAveragePooling2D(),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(128, activation='relu', kernel_regularizer=keras.regularizers.l2(0.001)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4, activation='softmax')
])

model.compile(optimizer=keras.optimizers.AdamW(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

def scheduler(epoch, lr):
    if epoch < 10:
        return lr
    else:
        return float(lr * 0.95)
```

```python
lr_scheduler = keras.callbacks.LearningRateScheduler(scheduler, verbose=1)

history = model.fit(
    train_dataset,
    validation_data=validation_dataset,
    epochs=30,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[early_stopping, lr_scheduler]
)

base_model.trainable = True
fine_tune_at = len(base_model.layers) // 2
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable = False

model.compile(optimizer=keras.optimizers.AdamW(learning_rate=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_fine = model.fit(
    train_dataset,
    validation_data=validation_dataset,
    epochs=15,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[early_stopping, lr_scheduler]
)

test_loss, test_accuracy = model.evaluate(test_dataset)
print(f"Test Accuracy: {test_accuracy:.2f}")

model.save('plant_lifecycle_classifier.keras')
```

Different changes:

Here follows the changes we made in our second iteration of the machine learning model:

1. Data Augmentation changes:
    - In the initial model we used the "ImageDataGenerator" to apply various types of augmentation, like shear, zoom and horizontal flipping. This led to more aggressive augmentation.
    - In the second iteration we used custom "Keras Sequential" augmentation with RandomFlip, RandomRotation(0.2) and RandomZoom(0.2). This strategy of augmentation is less aggressive.

2. Loading datasets:
    - The second iteration uses "image_dataset_from_directory" from tensorflow to load the datasets instead of the "ImageDataGenerator". This function creates "tf.data.Dataset" objects directly, which can be augmented, prefetched and mapped with greater control.

3. Prefetching for performance:
    - In the initial version we did not use prefetching.
    - In the second iteration we used prefetching so that the CPU can prepare the next batch while the GPU processes the current batch. This sped up the training time.

4. Batch size changes:
    - We changed the batch size from 32 to 16. The reason for this is so the model can be a bit more stable when training.

5. Model architecture and changes to layers:
    - In the second iteration we added an additional Batch Normalization layer before the output layer
    - The second iteration still kept it's dropout rate of 0.5.
    - We also added more layers and dropout to increase the complexity.

6. Fine Tuning:
    - In the initial version we fine tuned all of the layers of the base model after the initial training.
    - With the second iteration we just fine tuned the last half of the base model layers by unfreezing them.
    - This change helped balance the use of pre-trained knowledge with some more specific tuning tailored for our dataset.

7. Early stopping and learning rate scheduler:
    - In the second iteration we reduced the early stopping patience to only 10 epochs to prevent overfitting. This means that the training will stop earlier if the model is not improving.
    - We added a learning rate scheduler that reduced the learning rate by multiplying by 0.95 after 10 epochs. This helps improve convergence and slows down the learning rate as the training progresses, which helps the model settle into local minima more effectively.

8. Steps per epoch calculation:
    - In the initial version we used batches based on "ImageDataGenerator".
    - In the second version we defined "steps_per_epoch" and "validation_steps" manually by calculating based on the number of samples and the batch size.
    - We tried this approach to ensure that the entire dataset is utilized consistently. This can especially be useful if the dataset is not perfectly divisible by the batch size.

Results of the 2nd iteration:

The overall accuracy of the second iteration did improve quite a bit with the accuracy standing at 43% now.

Possible reasons for this inaccuracy:

- The model could possibly be overfitting. We can make this assumption because the validation accuracy fluctuated a lot and did not consistently improve.
- The reduced learning rate decrease might have been too aggressive during the fine tuning. This could be preventing the model from making substantial progress.

```
Epoch 13/30
110/110 ━━━━━━━━━━━━━━━━━━━━ 18s 163ms/step - accuracy: 0.4406 - loss: 1.7350 - val_accuracy: 0.4891 - val_loss: 1.4167 - learning_rate: 8.5737e-05

Epoch 14: LearningRateScheduler setting learning rate to 8.145062311086804e-05.
Epoch 14/30
110/110 ━━━━━━━━━━━━━━━━━━━━ 16s 143ms/step - accuracy: 0.4365 - loss: 1.7042 - val_accuracy: 0.6250 - val_loss: 1.2379 - learning_rate: 8.1451e-05

Epoch 1: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 1/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 48s 327ms/step - accuracy: 0.3116 - loss: 2.2382 - val_accuracy: 0.3995 - val_loss: 1.5680 - learning_rate: 1.0000e-05

Epoch 2: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 2/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 0s 286ms/step - accuracy: 0.3099 - loss: 2.21932024-10-27 21:13:26.679321: I tensorflow/core/framework/local_rendezvous.cc:404] Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
         [[{{node IteratorGetNext}}]]
110/110 ━━━━━━━━━━━━━━━━━━━━ 31s 287ms/step - accuracy: 0.3100 - loss: 2.2190 - val_accuracy: 0.3750 - val_loss: 1.7685 - learning_rate: 1.0000e-05

Epoch 3: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 3/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 310ms/step - accuracy: 0.3135 - loss: 2.2901 - val_accuracy: 0.4266 - val_loss: 1.6741 - learning_rate: 1.0000e-05

Epoch 4: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 4/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 31s 283ms/step - accuracy: 0.3472 - loss: 2.1480 - val_accuracy: 0.5000 - val_loss: 1.4170 - learning_rate: 1.0000e-05

Epoch 5: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 5/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 309ms/step - accuracy: 0.3481 - loss: 2.0726 - val_accuracy: 0.4538 - val_loss: 1.5813 - learning_rate: 1.0000e-05

Epoch 6: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 6/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 32s 289ms/step - accuracy: 0.3578 - loss: 2.0529 - val_accuracy: 0.2500 - val_loss: 2.1561 - learning_rate: 1.0000e-05

Epoch 7: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 7/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 312ms/step - accuracy: 0.3640 - loss: 2.0844 - val_accuracy: 0.4538 - val_loss: 1.5392 - learning_rate: 1.0000e-05

Epoch 8: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 8/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 31s 285ms/step - accuracy: 0.3703 - loss: 2.0789 - val_accuracy: 0.5000 - val_loss: 1.5176 - learning_rate: 1.0000e-05

Epoch 9: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 9/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 314ms/step - accuracy: 0.3756 - loss: 2.0081 - val_accuracy: 0.4837 - val_loss: 1.4981 - learning_rate: 1.0000e-05

Epoch 10: LearningRateScheduler setting learning rate to 9.999999747378752e-06.
Epoch 10/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 31s 284ms/step - accuracy: 0.3901 - loss: 2.0548 - val_accuracy: 0.3750 - val_loss: 1.5725 - learning_rate: 1.0000e-05
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 138ms/step - accuracy: 0.4238 - loss: 1.6421
Test Accuracy: 0.43
```

Third Iteration:

3rd Iteration of the Model Training script:

ModelTraining.py > ...

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications import EfficientNetB0
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import warnings
warnings.filterwarnings("ignore")

base_dir = 'Images'
train_dir = f'{base_dir}/train'
validation_dir = f'{base_dir}/validation'
test_dir = f'{base_dir}/test'

batch_size = 16

train_dataset = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
).repeat()

validation_dataset = tf.keras.utils.image_dataset_from_directory(
    validation_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
)

test_dataset = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
)

AUTOTUNE = tf.data.AUTOTUNE
train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)

num_train_samples = 1769
num_validation_samples = 376
steps_per_epoch = num_train_samples // batch_size
validation_steps = num_validation_samples // batch_size

data_augmentation = keras.Sequential([
    keras.layers.RandomFlip("horizontal"),
    keras.layers.RandomRotation(0.2),
    keras.layers.RandomZoom(0.2)
])
train_dataset = train_dataset.map(lambda x, y: (data_augmentation(x, training=True), y), num_parallel_calls=AUTOTUNE)
```

```python
base_model = EfficientNetB0(input_shape=(224, 224, 3), include_top=False, weights='imagenet')
base_model.trainable = False

model = keras.Sequential([
    base_model,
    keras.layers.GlobalAveragePooling2D(),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(128, activation='relu', kernel_regularizer=keras.regularizers.l2(0.001)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4, activation='softmax')
])

model.compile(optimizer=keras.optimizers.AdamW(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

history = model.fit(
    train_dataset,
    validation_data=validation_dataset,
    epochs=30,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[early_stopping]
)

base_model.trainable = True
fine_tune_at = len(base_model.layers) // 2
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable = False

model.compile(optimizer=keras.optimizers.AdamW(learning_rate=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_fine = model.fit(
    train_dataset,
    validation_data=validation_dataset,
    epochs=15,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[early_stopping]
)
```

```python
test_loss, test_accuracy = model.evaluate(test_dataset)
print(f"Test Accuracy: {test_accuracy:.2f}")

model.save('plant_lifecycle_classifier.keras')
```

Different changes:

1. Learning rate scheduler:

- In the third iteration we completely removed the learning rate scheduler. We did this to make the learning process a bit simpler, because the learning rate decrease in the second iteration might have been too aggressive.

Results of the 3rd iteration:

After removing the learning rate scheduler the model got an accuracy of 49%. Due to the model only improving with very minimal margins we decided to take a different approach and try to improve the preprocessing script.

```
Epoch 5/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 314ms/step - accuracy: 0.3093 - loss: 2.2688 - val_accuracy: 0.4348 - val_loss: 1.5611
Epoch 6/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 32s 287ms/step - accuracy: 0.3357 - loss: 2.1952 - val_accuracy: 0.3750 - val_loss: 1.5047
Epoch 7/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 316ms/step - accuracy: 0.3203 - loss: 2.2978 - val_accuracy: 0.4538 - val_loss: 1.5164
Epoch 8/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 31s 282ms/step - accuracy: 0.3537 - loss: 2.1635 - val_accuracy: 0.5000 - val_loss: 1.2844
Epoch 9/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 309ms/step - accuracy: 0.3504 - loss: 2.1295 - val_accuracy: 0.4565 - val_loss: 1.4873
Epoch 10/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 315ms/step - accuracy: 0.3954 - loss: 2.1167 - val_accuracy: 0.6250 - val_loss: 1.0040
Epoch 11/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 38s 343ms/step - accuracy: 0.3889 - loss: 2.0611 - val_accuracy: 0.4728 - val_loss: 1.4563
Epoch 12/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 318ms/step - accuracy: 0.3840 - loss: 2.0362 - val_accuracy: 0.3750 - val_loss: 1.5441
Epoch 13/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 38s 344ms/step - accuracy: 0.3989 - loss: 2.0564 - val_accuracy: 0.4864 - val_loss: 1.4333
Epoch 14/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 311ms/step - accuracy: 0.4131 - loss: 1.9724 - val_accuracy: 0.6250 - val_loss: 1.2280
Epoch 15/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 38s 345ms/step - accuracy: 0.3980 - loss: 1.9241 - val_accuracy: 0.5027 - val_loss: 1.4144
24/24 ━━━━━━━━━━━━━━━━━━━━ 4s 147ms/step - accuracy: 0.4418 - loss: 1.5221
Test Accuracy: 0.49
```

Fourth Iteration:

2nd Iteration of the Preprocessing script:

PreProcessing.py > ...

```
import tensorflow as tf

base_dir = 'Images'
train_dir = f'{base_dir}/train'
validation_dir = f'{base_dir}/validation'
test_dir = f'{base_dir}/test'

train_dataset = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

validation_dataset = tf.keras.utils.image_dataset_from_directory(
    validation_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

test_dataset = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

normalization_layer = tf.keras.layers.Rescaling(1.0 / 255)

data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip('horizontal'),
    tf.keras.layers.RandomRotation(0.2),
    tf.keras.layers.RandomZoom(0.2),
    tf.keras.layers.RandomBrightness(0.2),
    tf.keras.layers.RandomContrast(0.2)
])

AUTOTUNE = tf.data.AUTOTUNE
train_dataset = train_dataset.map(lambda x, y: (normalization_layer(data_augmentation(x, training=True)), y), num_parallel_calls=AUTOTUNE)

validation_dataset = validation_dataset.map(lambda x, y: (normalization_layer(x), y), num_parallel_calls=AUTOTUNE)
test_dataset = test_dataset.map(lambda x, y: (normalization_layer(x), y), num_parallel_calls=AUTOTUNE)

train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)

print("Train dataset:", train_dataset)
print("Validation dataset:", validation_dataset)
print("Test dataset:", test_dataset)
```

Different changes:

1. The saving of the preprocessed images:
    - In the second iteration we decided to not save the images to the disk. We added this change to improve efficiency, thereby speeding up the overall workflow.

2. Applying augmentation and normalization:
    - In the initial script data augmentation was only applied when saving the training images, and the images were only normalized after augmentation.
    - However in the second script we applied dynamic augmentation and normalization. This augmentation style is model flexible, providing different augmented samples during each training epoch.
    - We reduced the augmentation parameters to be less aggressive:

- RandomRotation(0.2)
- RandomZoom(0.2)
- RandomBrightness(0.2)
- RandomContrast(0.2)

3. Dataset prefetching:
    - We added prefetching in the second iteration to improve the training speed, which ultimately enhanced the model's performance.

Results of the 4th iteration:

Applying a few changes to the preprocessing script did help improve the model's accuracy to 56%.

```
Epoch 7/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 313ms/step - accuracy: 0.4710 - loss: 1.5485 - val_accuracy: 0.5076 - val_loss: 1.3674
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 313ms/step - accuracy: 0.4710 - loss: 1.5485 - val_accuracy: 0.5076 - val_loss: 1.3674
Epoch 8/15
Epoch 8/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 314ms/step - accuracy: 0.4746 - loss: 1.5710 - val_accuracy: 0.5267 - val_loss: 1.3588
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 314ms/step - accuracy: 0.4746 - loss: 1.5710 - val_accuracy: 0.5267 - val_loss: 1.3588
Epoch 9/15
Epoch 9/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 314ms/step - accuracy: 0.4560 - loss: 1.5600 - val_accuracy: 0.5305 - val_loss: 1.3535
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 314ms/step - accuracy: 0.4560 - loss: 1.5600 - val_accuracy: 0.5305 - val_loss: 1.3535
Epoch 10/15
Epoch 10/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 321ms/step - accuracy: 0.4768 - loss: 1.4773 - val_accuracy: 0.5420 - val_loss: 1.3410
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 321ms/step - accuracy: 0.4768 - loss: 1.4773 - val_accuracy: 0.5420 - val_loss: 1.3410
Epoch 11/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 322ms/step - accuracy: 0.4547 - loss: 1.5374 - val_accuracy: 0.5534 - val_loss: 1.3380
Epoch 12/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 322ms/step - accuracy: 0.4547 - loss: 1.5374 - val_accuracy: 0.5534 - val_loss: 1.3380
Epoch 12/15
Epoch 12/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 309ms/step - accuracy: 0.4682 - loss: 1.5334 - val_accuracy: 0.5534 - val_loss: 1.3338
Epoch 13/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 313ms/step - accuracy: 0.4995 - loss: 1.4296 - val_accuracy: 0.5382 - val_loss: 1.3310
Epoch 14/15
Epoch 14/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 35s 316ms/step - accuracy: 0.5095 - loss: 1.4375 - val_accuracy: 0.5496 - val_loss: 1.3178
Epoch 15/15
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 312ms/step - accuracy: 0.4921 - loss: 1.4595 - val_accuracy: 0.5496 - val_loss: 1.3184
110/110 ━━━━━━━━━━━━━━━━━━━━ 34s 312ms/step - accuracy: 0.4921 - loss: 1.4595 - val_accuracy: 0.5496 - val_loss: 1.3184
17/17 ━━━━━━━━━━━━━━━━━━━━ 2s 143ms/step - accuracy: 0.5529 - loss: 1.3357
Test Accuracy: 0.56
```

Final Model training and Preprocessing scripts:

After implementing a couple of other techniques, we were unable to produce a model that is more accurate than this one. This model gave us an accuracy of 56% and the proof of the results are mentioned above in the 4th iteration results.

Model training script:

ModelTraining.py > ...

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.applications import EfficientNetB0
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import warnings
warnings.filterwarnings("ignore")

base_dir = 'Images'
train_dir = f'{base_dir}/train'
validation_dir = f'{base_dir}/validation'
test_dir = f'{base_dir}/test'

batch_size = 16

train_dataset = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
).repeat()

validation_dataset = tf.keras.utils.image_dataset_from_directory(
    validation_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
)

test_dataset = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    image_size=(224, 224),
    batch_size=batch_size,
    label_mode='categorical'
)

AUTOTUNE = tf.data.AUTOTUNE
train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)

num_train_samples = 1769
num_validation_samples = 376
steps_per_epoch = num_train_samples // batch_size
validation_steps = num_validation_samples // batch_size

data_augmentation = keras.Sequential([
    keras.layers.RandomFlip("horizontal"),
    keras.layers.RandomRotation(0.2),
    keras.layers.RandomZoom(0.2)
])
train_dataset = train_dataset.map(lambda x, y: (data_augmentation(x, training=True), y), num_parallel_calls=AUTOTUNE)
```

```python
base_model = EfficientNetB0(input_shape=(224, 224, 3), include_top=False, weights='imagenet')
base_model.trainable = False

model = keras.Sequential([
    base_model,
    keras.layers.GlobalAveragePooling2D(),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(128, activation='relu', kernel_regularizer=keras.regularizers.l2(0.001)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4, activation='softmax')
])

model.compile(optimizer=keras.optimizers.AdamW(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

history = model.fit(
    train_dataset,
    validation_data=validation_dataset,
    epochs=30,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[early_stopping]
)

base_model.trainable = True
fine_tune_at = len(base_model.layers) // 2
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable = False

model.compile(optimizer=keras.optimizers.AdamW(learning_rate=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_fine = model.fit(
    train_dataset,
    validation_data=validation_dataset,
    epochs=15,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[early_stopping]
)
```

```python
test_loss, test_accuracy = model.evaluate(test_dataset)
print(f"Test Accuracy: {test_accuracy:.2f}")

model.save('plant_lifecycle_classifier.keras')
```

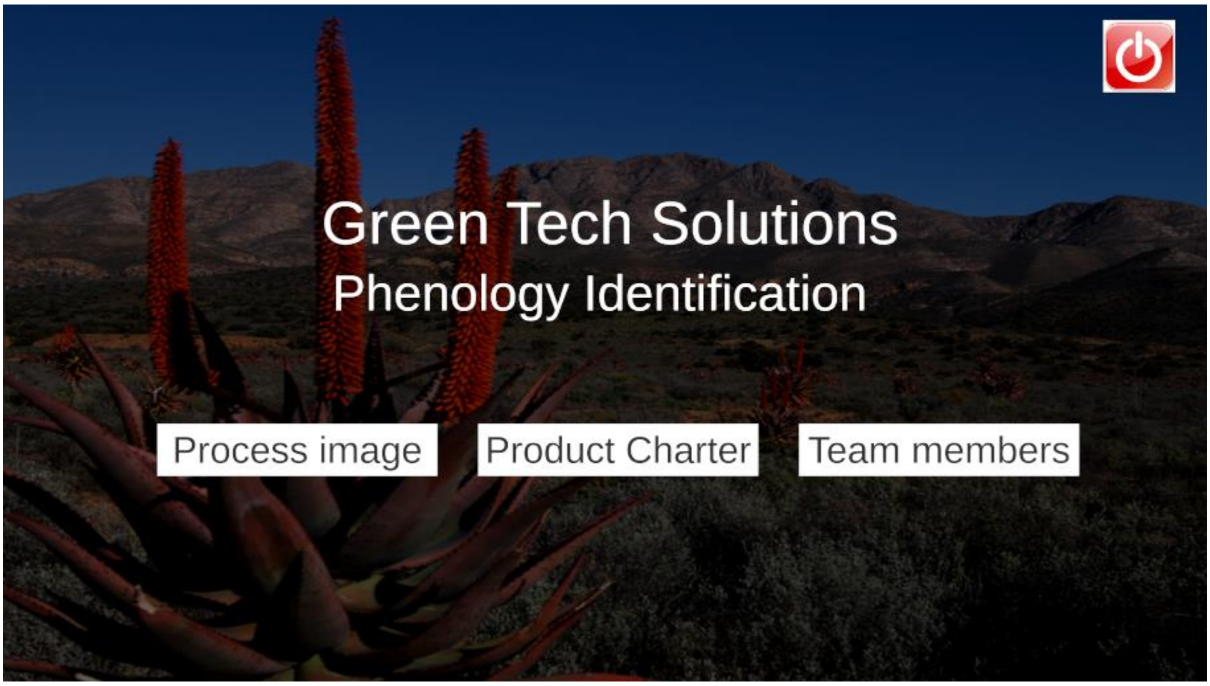
Preprocessing script:

PreProcessing.py > ...

```python
import tensorflow as tf

base_dir = 'Images'
train_dir = f'{base_dir}/train'
validation_dir = f'{base_dir}/validation'
test_dir = f'{base_dir}/test'

train_dataset = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

validation_dataset = tf.keras.utils.image_dataset_from_directory(
    validation_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

test_dataset = tf.keras.utils.image_dataset_from_directory(
    test_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='categorical'
)

normalization_layer = tf.keras.layers.Rescaling(1.0 / 255)

data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip('horizontal'),
    tf.keras.layers.RandomRotation(0.2),
    tf.keras.layers.RandomZoom(0.2),
    tf.keras.layers.RandomBrightness(0.2),
    tf.keras.layers.RandomContrast(0.2)
])

AUTOTUNE = tf.data.AUTOTUNE
train_dataset = train_dataset.map(lambda x, y: (normalization_layer(data_augmentation(x, training=True)), y), num_parallel_calls=AUTOTUNE)

validation_dataset = validation_dataset.map(lambda x, y: (normalization_layer(x), y), num_parallel_calls=AUTOTUNE)
test_dataset = test_dataset.map(lambda x, y: (normalization_layer(x), y), num_parallel_calls=AUTOTUNE)

train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
validation_dataset = validation_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)

print("Train dataset:", train_dataset)
print("Validation dataset:", validation_dataset)
print("Test dataset:", test_dataset)
```

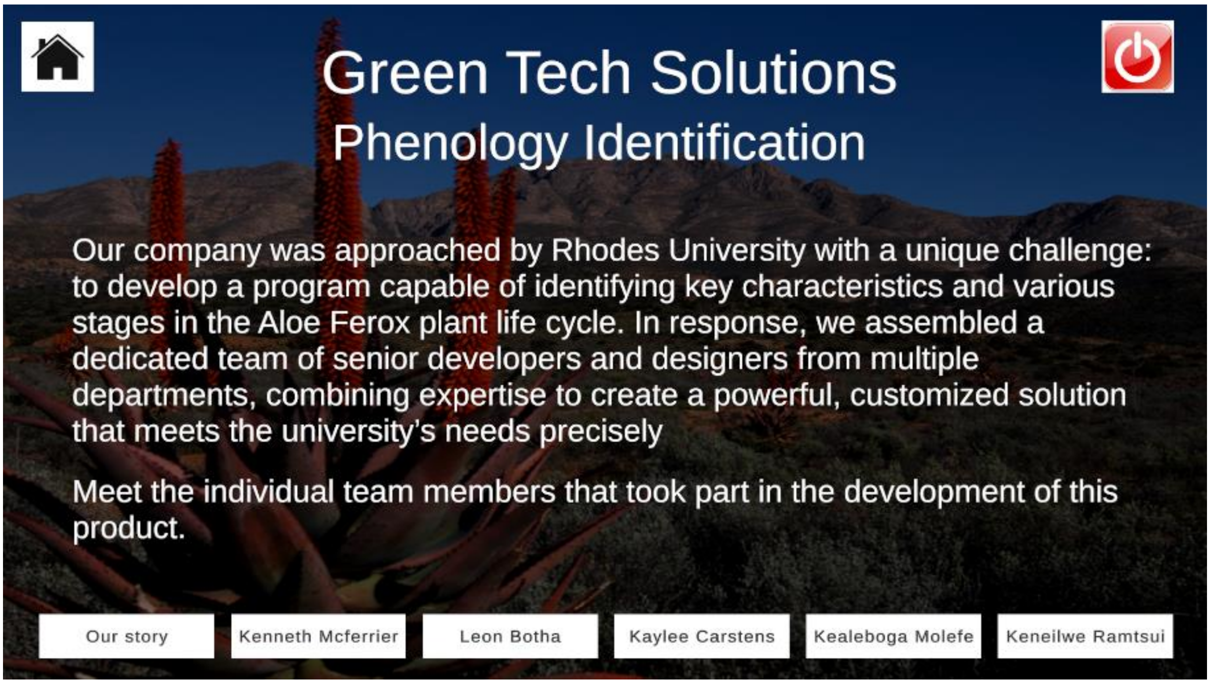
UI Design

<u>Homepage:</u>



<u>Team members page:</u>

After clicking on the "Team members" button you get relocated to our page that contains our group story and different buttons for our individual team members.

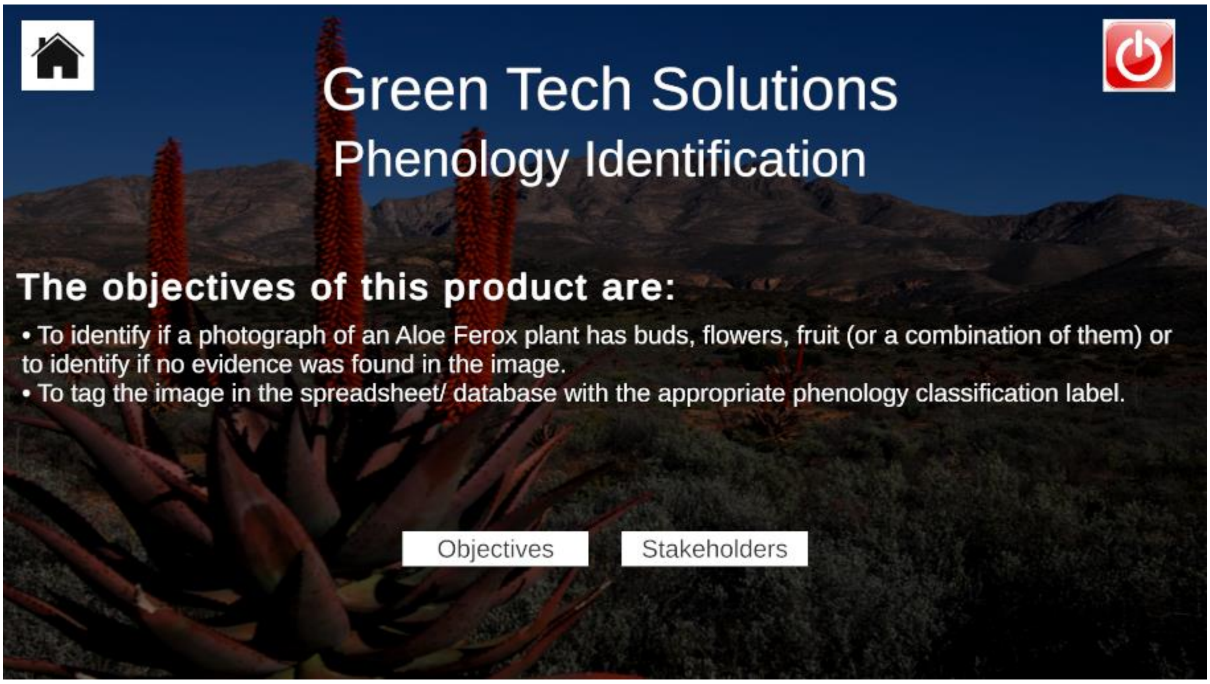


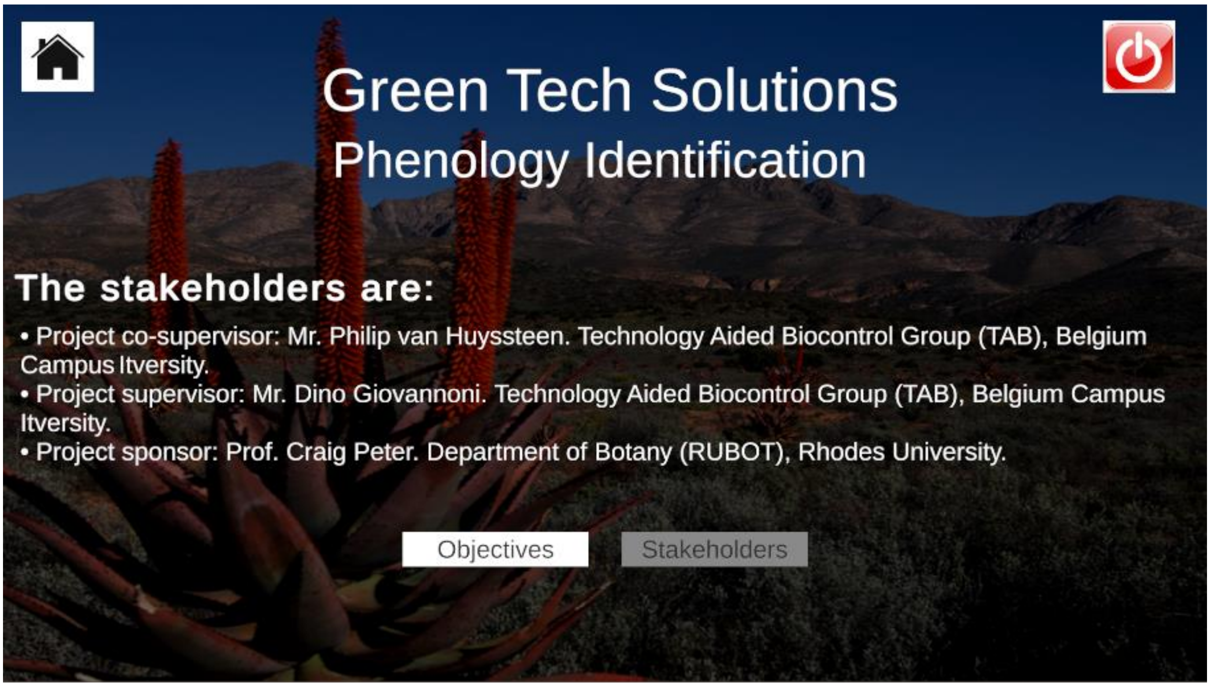
Team member Information:

When clicking on an individual team member's button, you are shown the different roles and responsibilities of that member.

Product charter:

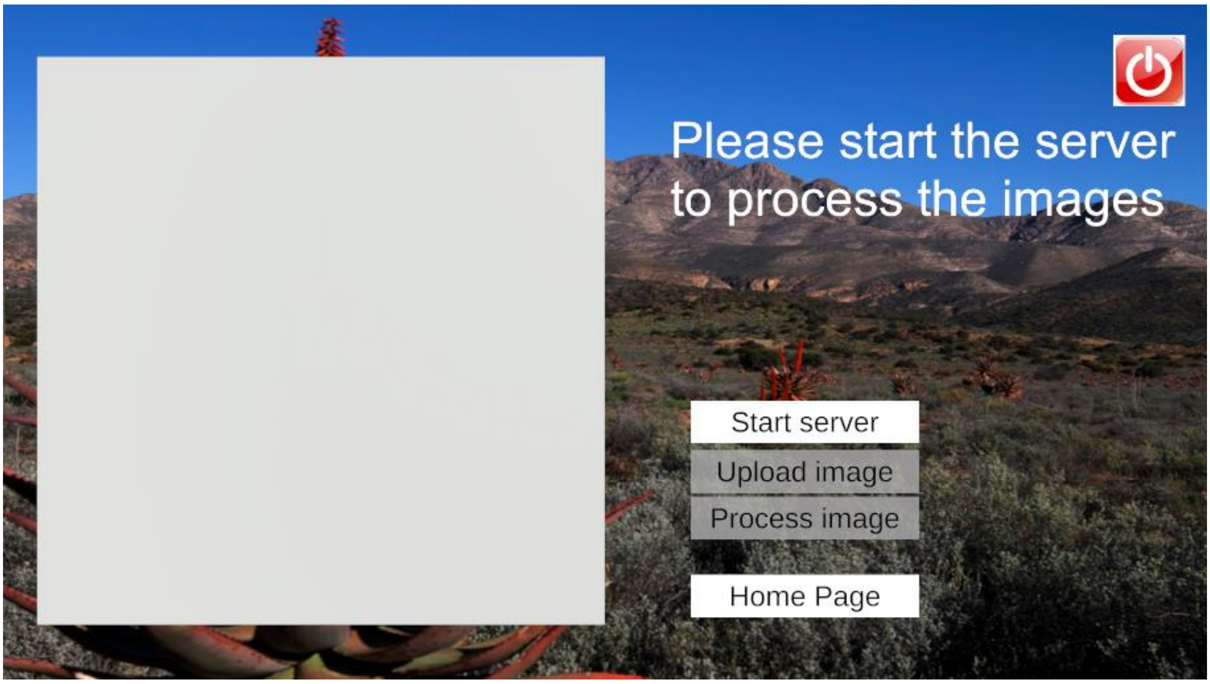
After going back to the homepage using the little house button and clicking on the product charter button you get transported to this page. This page just showcases the objective that this product wants to achieve and also the stakeholders in this product.



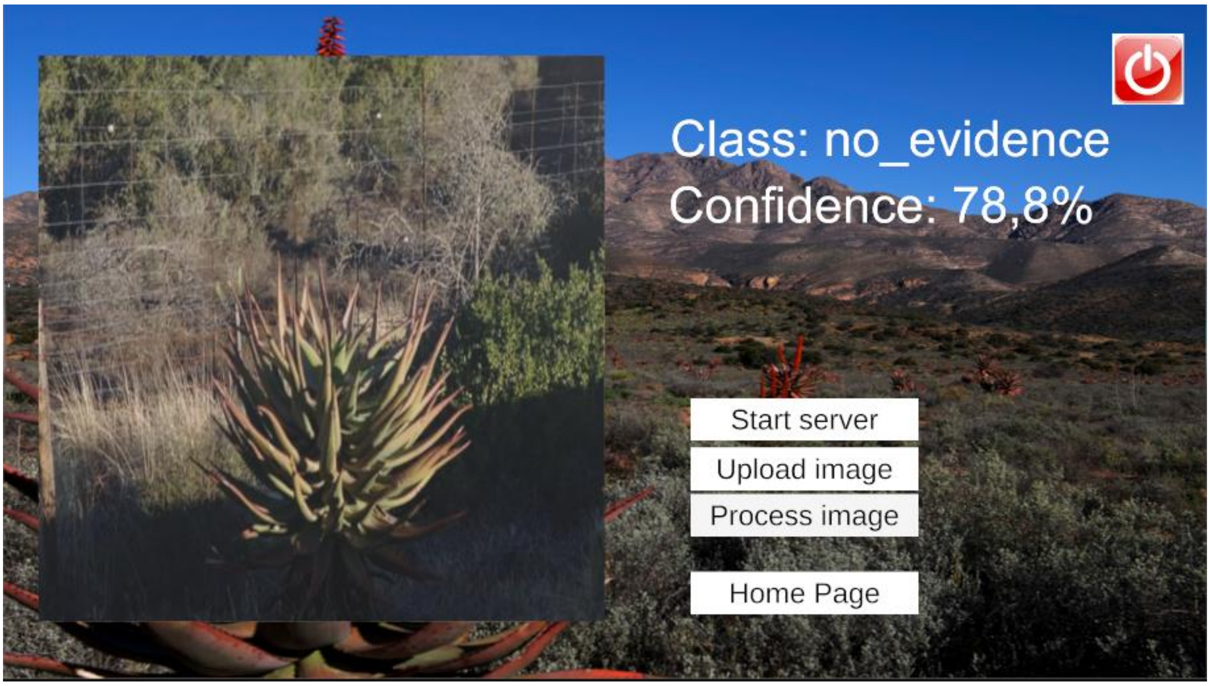


<u>Image processing page:</u>

Now moving on to the main attraction of our product, the image processing page. Here you can see very clear instructions on how to start this program. First you need to click the "Start server" button then wait 10 seconds for the UI to fetch the model. After that just click on the "Upload image" button then a popup will appear where you have to choose the picture you want to be classified.



After clicking on the “Process image” button you will get your results as follows.



Conclusion

Though we have a working product with a moderate accuracy rate, this is still a very promising step forward in the field of plant research. This product proves that the automation of identifying plant phenology is possible and also very feasible and is worth considering for the use of research.