UNIVERSITI TEKNOLOGI MARA

SEMESTER	**:**	**MARCH 2024 – AUG 2024**
COURSE	**:**	**IMAGE PROCESSING**
COURSE CODE	**:**	**CSC566**
PROJECT TITLE	**:**	**INNOVATIVE IMAGE PROCESSING FOR EARLY DETECTION OF CORN LEAF DISEASE**

Group Name:	Group 2
Name, Student ID:	1. AHMAD SUFFIAN BIN MUHAMMAD SHAHRIL (2021494474)
	2. LINA SYAZANI ILIYA BINTI NAZARUDIN (2021454602)
	3. NORAMIRA ATHIRAH BINTI NOR AZMAN (2021899762)
	4. SYAZREEL NAJMI BIN MOHAMAD SAHRUM NIZAM (2021459038)

TEMPLATE FOR PROJECT GRADING

Each group should complete and print this sheet. The reports along with datasets and prototype should be made available in Google Drive.

ITEMS	Percentage	MARKS
PRESENTATION: 1. Completeness of prototype 2. Content of presentation 3. Delivery skills 4. Question & Answer	**10%**	
REPORT: 1. Introduction & project significance 2. Data collection a) Training dataset b) Testing dataset 3. Flowchart of prototype 4. Deep learning architecture a) Input b) Feature extraction layer: convolution and pooling c) Classification layer: fully-connected d) Output 5. Results of test dataset a) Learning rate and accuracy rate b) Error rate 6. Performance Evaluation a) Accuracy/F1 Score b) Recall c) Precision	**30%**	
TOTAL MARKS	**40%**	

Table of Content

1.0 Introduction

Corn leaf disease detection is a part of crop health monitoring. It involves continuously observing and analyzing crops to identify early signs of stress, disease, or nutritional deficiencies. This proactive approach helps ensure timely interventions to maintain optimal crop health. In contemporary agriculture, crop health monitoring is crucial for promptly identifying diseases, stress, and nutrient deficits in crops (Omia et al., 2023). Utilizing a range of methods and tools, this procedure evaluates the health of plants and makes sure that any negative impacts are taken care of quickly. By seeing problems like pest infestations, water stress, and nutrient imbalances early on, effective crop health monitoring enables farmers to make the necessary corrections.

Furthermore, monitoring crop health is crucial to boost agricultural production. Fast action in problem identification can result in major gains in crop quality and yields, as well as efficient use of resources like pesticides, fertilizers, and water. Farmers may lower the risk of crop failure, limit financial losses, and improve overall food security by recognizing and resolving problems early on. Moreover, by using sustainable farming methods, this strategy reduces the chance of crop failure and associated financial losses while improving overall food security (Shafi et al., 2019).

Thus, the purpose of this project is to build advanced image processing methods for efficient crop health monitoring, specifically focusing on classifying corn leaf diseases. Using a Kaggle dataset, the project categorizes diseases affecting corn leaves by leveraging unique visual characteristics for accurate classification. Modern image processing techniques identify diseases like Blight, Gray Leaf Spot, and Common Rust, comparing them to healthy leaves to improve detection accuracy. This approach supports early diagnosis, enhances agricultural yields, improves resource management, and promotes environmentally friendly farming practices.

2.0 Objectives

The following are project objectives:

1) To identify suitable techniques for image processing in classifying corn leaf diseases.
2) To develop a classification model to easily classify corn leaf diseases.
3) To evaluate the performance of the classification model in classifying corn leaf diseases.

3.0 Project Significance

This project holds significant value and offers various benefits:

1. Improved Disease Management: Enables early detection of corn leaf diseases, ensuring timely intervention and accurate classification for effective treatment.
2. Economic Benefits: Reduces costs associated with pesticide use and crop losses, while increasing overall farm productivity and profitability through higher yields.
3. Environmental Impact: Promotes sustainable agriculture by reducing pesticide use through targeted application strategies and encouraging precise disease management practices, thereby minimizing environmental impact and promoting sustainable farming practices.
4. Technological Advancement: Integrates advanced image processing and deep learning technologies, fostering innovation in agriculture and enhancing scalability across different crops and regions.
5. Global Food Security: Enhances corn health and yield, contributing to global food security by supporting sustainable agricultural practices and benefiting smallholder farmers with improved access to agricultural advancements.

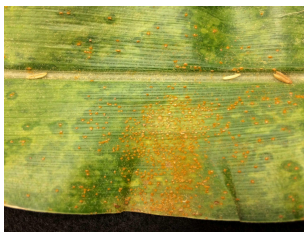
4.0 Data Collection

In this project, corn leaf disease classification is the main focus to be enhanced with image processing technologies. For the sole purpose of this project, the dataset that has been chosen is an image dataset of a specific crop which is corn. The dataset is titled “Corn or Maize Leaf Disease Dataset” which was obtained from the Kaggle website. This dataset provides different disease classes of corn leaf images that can be used for classification. For the convenience of access, the dataset can be found at the following link: Corn or Maize Leaf Disease Dataset.

In this dataset, there are a total of four different classes that indicate the disease or health of the corn plants [see Table 1]. This makes up a total of 4,188 images for this dataset.

Table 1 Total Number of Images for Each Class

Class	Sample	Total images
Blight		1,306
Gray Leaf Spot		574

Common Rust		1,146
Healthy		1,162

The dataset chosen is highly suitable for this project and prioritizes corn leaf disease detection for several reasons. First and foremost, the dataset focuses on specifically common diseases that usually occur in corn leaf, such as common rust, gray leaf spot, and blight. The production of corn is seriously threatened by these diseases, which makes efficient monitoring and early detection important for controlling crop health. On top of that, the dataset, which consists of 4,188 images categorized into four classes, offers a thorough depiction of the visual representation that is linked to these diseases. In addition, the number of images per class for this dataset is enough to create a robust deep-learning model that can yield great accuracy for classification.

Furthermore, this dataset aligns with the project's primary goal of corn leaf disease classification because it contains images of corn leaves in healthy and diseased states. From this project, the findings should be possibly applicable to actual agricultural operations due to the alignment with the objective. In conclusion, the “Corn or Maize Leaf Disease Dataset” from Kaggle is an ideal dataset to be chosen for constructing a crop health monitoring system by employing image processing technology.

5.0 Flowchart of Prototype

This section presents flowcharts for developing a corn leaf disease classification model using image processing techniques. It covers the main, image processing, and image enhancement flowcharts, detailing three key methods: Adaptive Histogram Equalization (AHE), Canny Edge Detection, and Texture Extraction.

a) Main Flowchart

The flowchart (Figure 1) outlines the development of a corn leaf disease classification system. The process starts with load dataset collection (refer to Appendix A) and undergoes image processing of a dataset of diseased corn leaf images. The data is then split into training (85%), validation (7.5%), and testing (7.5%) datasets. The training dataset is used to train a CNN model, which is then validated with the validation dataset, iterating to find optimal hyperparameters until optimal performance is achieved. Finally, the trained and validated model is evaluated with the testing dataset to ensure its accuracy, precision, recall, and F1 score in classifying diseases.

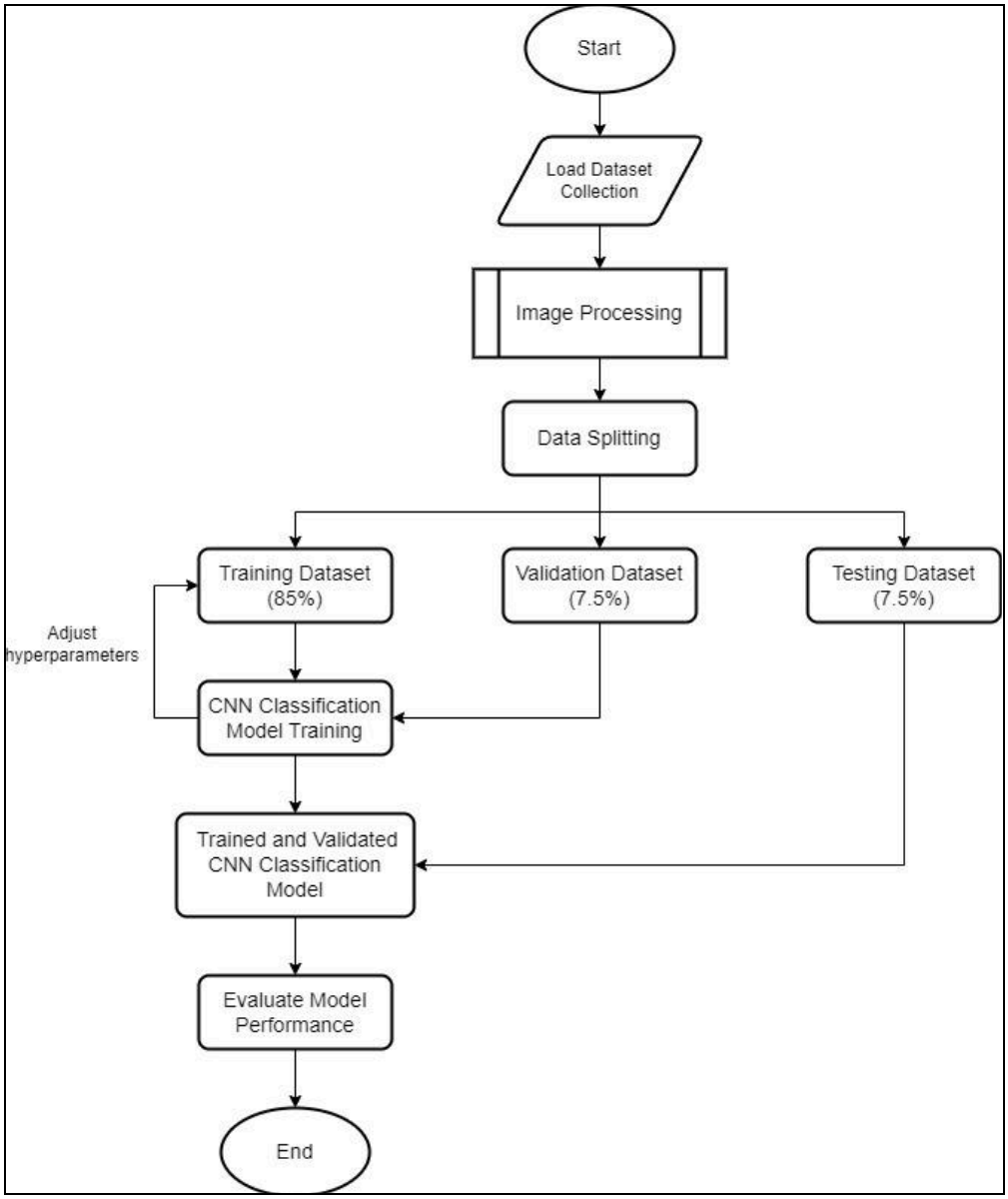
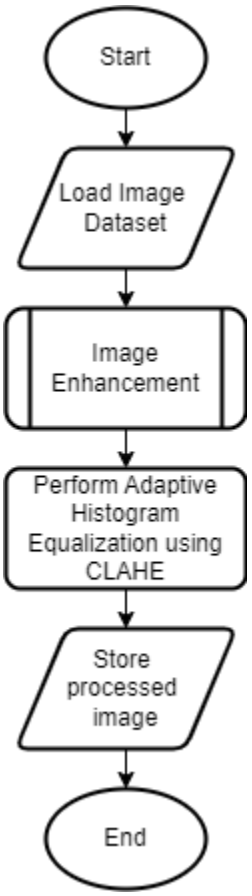


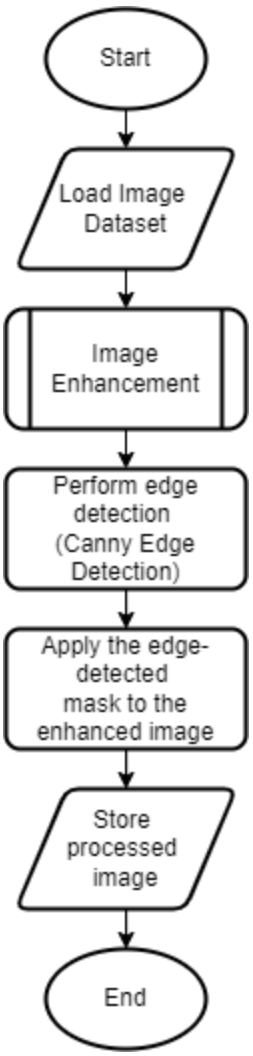
Figure 1 Prototype's Flowchart

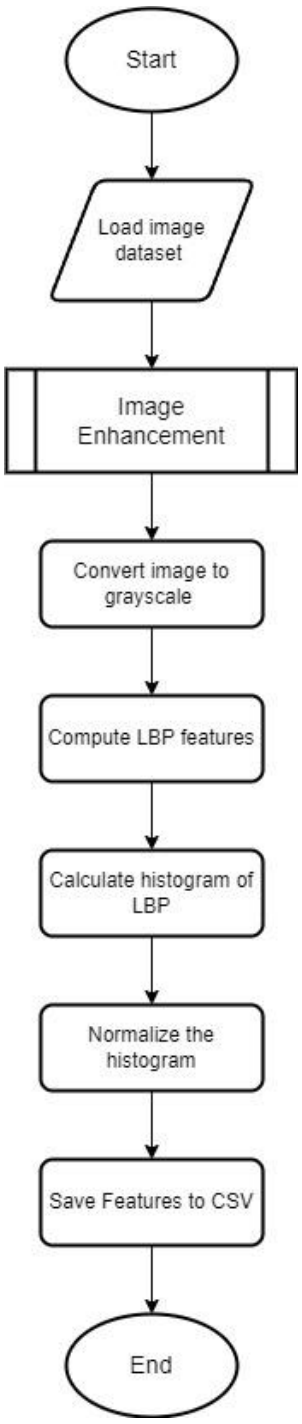
b) Image Processing Flowchart

The following figures illustrate the three image processing techniques through flowcharts. Refer to Appendix B for detailed code implementations for each technique.

Table 2 Image Processing Flowchart Explanation

Technique	Flowchart	Description
Technique 1: AHE	 **Figure 2** Technique 1 Flowchart	Based on Figure 2, the flowchart shows the imaging technique using AHE on Contrast Limited AHE (CLAHE). The method begins with loading an image dataset, followed by image enhancement techniques to prepare the images for subsequent processing. The key operation is AHE with CLAHE, which adjusts the contrast of the images locally rather than globally, preserving details in different regions of the images. This makes it especially effective in images with varying lighting conditions or where important details are covered up by shadows or highlights. After the CLAHE is applied, the processed photos are saved, indicating the completion of the process. This technique is especially beneficial for improving the visibility of details in images with different lighting conditions.

Technique 2: Canny Edge Detection	 **Figure 3** Technique 2 Flowchart	The flowchart (Figure 3) illustrates the process for preprocessing images in a dataset. It begins with loading the dataset, where each image is read from the designated directory. Each image then will undergo image enhancement processes. After image enhancement, Canny Edge Detection is applied to emphasize the edges. This edge-detected image is then applied to the enhanced image to highlight the edges of the image. Then, the process includes saving the applied Canny Edge Detection into a new directory to be used later for building the image classification model.

Technique 3: Texture Extraction	 **Figure 4** Technique 3 Flowchart	The flowchart (Figure 4) outlines the process for extracting texture features from an image dataset. It begins with loading the image dataset, followed by an image enhancement step to improve the quality of the images. Next, the images are converted to grayscale to simplify the analysis. The grayscale images are then used to compute Local Binary Pattern (LBP) features, which capture the texture information. A histogram of these LBP features is calculated to quantify their distribution. This histogram is subsequently normalized to ensure that the sum of its values equals 1, making it suitable for further analysis. After normalization, the process includes saving the computed and normalized feature values into CSV files to preserve the extracted texture information for building the classification model.

c) Image Enhancement Flowchart

In Figure 5, the flowchart illustrates a multi-step image enhancement method. It begins with color correction, adjusting the image's color space for better processing and natural appearance. Following that, contrast enhancement is used to strengthen the differentiation between various areas of the image, making details easier to see. The image is then transformed back to a numpy array in RGB color space, ready for further processing. To maintain size consistency, the image is padded to make it square and then enlarged to the appropriate dimensions. These steps enhance visual quality and prepare the image for further processing. See Appendix C for code details.

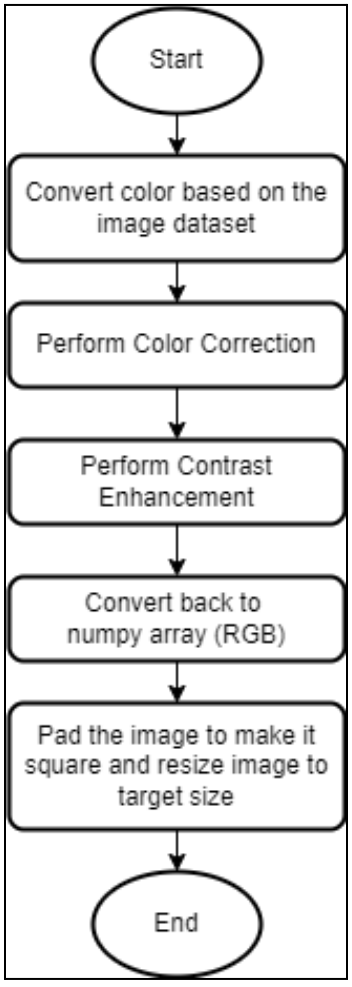


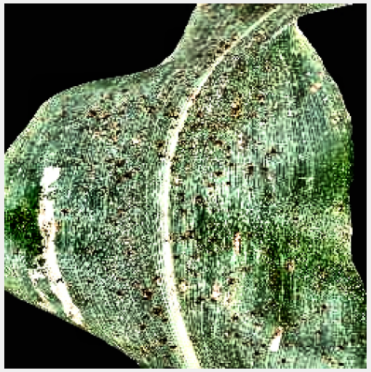
Figure 5 Image Enhancement Flowchart

6.0 Result of Image Processing

This section will showcase sample images processed through each image processing technique. Each technique will be demonstrated with multiple images, highlighting the changes at different stages of the process.

a) Sample of Image Processing Technique 1 - AHE

Table 3 Sample of Image Processing Technique 1 - AHE

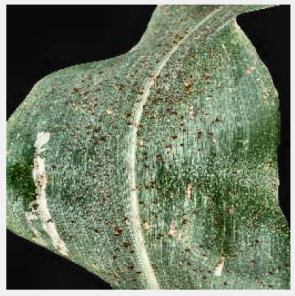
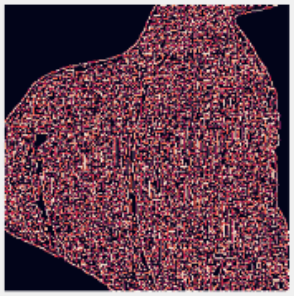
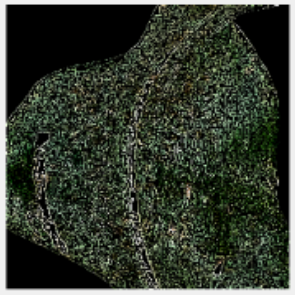
Original	Color Correction	Contrast Enhancement	Adaptive Histogram Equalized
			

Based on the sample of the AHE technique, starts with the original image, the leaf with some details visible, but the overall image is muted and lacks contrast, making it difficult to detect smaller details. Following that, the "Color Correction technique adjusts the leaf's colors to make them appear more natural and accurate. This step improves the color balance, brightening the greens and making the overall image more visually pleasing. However, despite

the better colors, the image still lacks appropriate contrast. Next, in the "Contrast Enhancement" stage, the contrast between different sections of the image is raised, making details more visible. This step improves the visibility of textures and characteristics on the leaf, resulting in a sharper and more detailed view than earlier stages. Finally, the image "Adaptive Histogram Equalized" displays the use of AHE. This approach improves contrast enhancement by changing it locally in different areas of the image. As a result, the details in both the darker and brighter parts of the leaf become more detailed, providing a more balanced perspective. The AHE stage successfully exposes the leaf's rich patterns and textures, resulting in the most detailed and visually interesting updated version of the image.

b) Sample of Image Processing Technique 2 - Canny Edge Detection

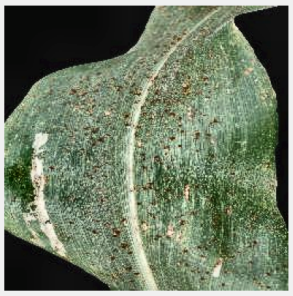
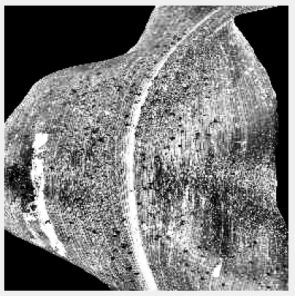
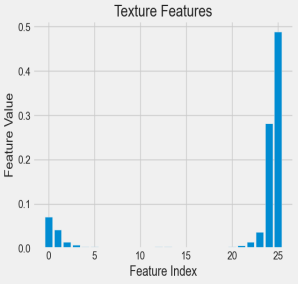
Table 4 Sample of Image Processing Technique 2 - Canny Edge Detection

Original	Color Correction	Contrast Enhancement	Canny Edge Detection	Applied Canny Edge Detection
				

Based on the Canny Edge Detection technique's sample, begins with the "Original" image, which shows the leaf that is not that clear in detail due to the muted contrast and color. The "Color Correction" stage adjusts the colors to make it appear more natural and precise. The greens become more vibrant, adding to the overall visual appearance of the image, although the contrast remains relatively low. The contrast is then increased to highlight the details of the leaf in the "Contrast Enhancement" stage. In comparison to the original and color-corrected images, this stage greatly improves the visibility of textures and details on the leaf by providing a clearer and more detailed view. The "Canny Edge Detection" stage displays the results of the algorithm that locates the edges in the image. The highlighted edges that outline the leaf's features and boundaries serve as a representation of this. By highlighting the edges, the Canny Edge Detection efficiently captures the complex details and structures. Lastly, the detected edges are overlaid on the enhanced image in the "Applied Canny Edge Detection" image. The leaf's textures and structural details are emphasized in these combined results, which simplifies feature analysis. The clearly defined edges improve the overall comprehension of the image's structure and offer an in-depth overview of the morphology of the leaf.

c) Sample of Image Processing Technique 3 - Texture Extraction

Table 5 Sample of Image Processing Technique 3 - Texture Extraction

Original	Color Correction	Contrast Enhancement	Gray Scale	Extracted Features Values (Histogram)
				

Based on the sample of the Texture Extraction technique, starts with the original image, where the leaf details are visible but muted and lacking contrast. Following that, the Color Correction technique adjusts the leaf's colors to make them appear more natural and accurate. This step improves the color balance, brightening the greens and making the overall image more visually pleasing but still lacking contrast. Next, in the Contrast Enhancement stage, the contrast is increased, making details and textures more visible and resulting in a sharper image. The image is then converted to Gray Scale, simplifying it by removing color and focusing on light and dark intensities, which is essential for texture analysis. Finally, the Extracted Features Values (Histogram) show the distribution of LBP features from the grayscale image. This histogram captures the texture information of the leaf, with peaks and distributions indicating various texture patterns, which are crucial for classifying the leaf's condition. The histogram is normalized, ensuring the sum of its values equals 1, making it suitable for further analysis and comparison

7.0 Dataset Splitting

The dataset is divided into three subsets: training, validation, and testing, as described in the following section. For the specific code implementation, refer to Appendix D.

a) Training Dataset

The training dataset includes 85% of the 4,188 photos in the "Corn or Maize Leaf Disease Dataset," which is equivalent to around 3,560 images. These images were carefully chosen using the train_test_split function to illustrate the four categories: 'Common Rust,' 'Gray Leaf Spot,' 'Blight,' and 'Healthy'. The primary function of having a training dataset is to provide a comprehensive set of examples for the model to learn from which can thus identify and learn the unique patterns and characteristics associated with each disease category.

To increase image quality and consistency, the training data is preprocessed using techniques such as color correction and contrast enhancement. Following this, additional techniques like AHE, Canny Edge Detection, and Texture Extraction further refine the images. AHE enhances contrast to improve the visibility of details, while Canny Edge Detection highlights disease boundaries. Feature Extraction isolates key image attributes essential for distinguishing disease classes. These methods collectively enhance dataset robustness, boosting model accuracy. The trained Convolutional Neural Network (CNN) then learns distinctive disease patterns, facilitated by a diverse training set, ensuring reliable performance on new, unseen images in practical settings.

b) Validation Dataset

The validation dataset accounts for 7.5% of the 4,188 photographs in the "Corn or Maize Leaf Disease Dataset," or approximately 314 images. This subset of data is critical for adjusting the model's hyperparameters and comparing multiple models during the training phase. Unlike the training dataset, the validation dataset helps in determining how well the model learns and whether it can generalize effectively to new data.

Validation involves iterative cycles of training on the training data and testing on the validation data. This iterative process continues until achieving satisfactory performance, which in this case is a 98% accuracy score. It ensures that the model is both accurate and robust, ready to effectively classify new and unknown images.

c) Testing Dataset

The testing dataset includes the remaining 7.5% of the total images or around 314 images. This subset is maintained separately from the training data to ensure a fair evaluation of the model's performance. The testing dataset is used to ensure that the model is accurate and effective at classifying corn leaf diseases. It contains photographs from the same four classes: Common Rust, Gray Leaf Spot, Blight, and Healthy Leaves. By evaluating the model on this independent dataset, we can confirm that it has not overfitted to the training data and can correctly categorize new, previously unseen images. The metrics collected during testing, such as accuracy, precision, recall, and F1 score, provide important insights into the model's reliability and potential for agricultural application.

8.0 Deep Learning Architecture

a) Input

The input for the model has been preprocessed using image enhancement techniques and three distinct image processing techniques, which are AHE, Canny Edge Detection, and Texture Extraction. The first two techniques will give input to the model as a 256x256x3 shape (RGB images) of corn plants in four conditions: Blight, Gray Leaf Spot, Common Rust, and Healthy. While for Technique 3, it gives input of vector to the model.

b) Feature Extraction Layer: Convolution and Pooling

In this project, three different image processing techniques are being utilized: AHE, Texture Extraction, and Canny Edge Detection. The convolutional base for Techniques 1 and 2, is defined with a common pattern of stacked Conv2D and MaxPooling2D layers. These layers work together to perform feature extraction on the input images. While for Technique 3, as mentioned the input to the model is vectorized, it does not retain the spatial structure of the image. Hence, it requires a different model architecture due to the nature of its input feature, making a fully connected architecture more suitable. Hence, there is no convolutional and pooling layer for Technique 3.

As in Figure 6, each Conv2D layer applies a set of filters to the input, producing feature maps that highlight various aspects of the input image such as edges, textures, and patterns. The MaxPooling2D layers then downsample these feature maps, reducing their spatial dimensions while retaining the most important information.

```
layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape, name='Conv1'),
layers.MaxPooling2D((2, 2), name='MaxPool1'),
layers.Conv2D(64, (3, 3), activation='relu', name='Conv2'),
layers.MaxPooling2D((2, 2), name='MaxPool2'),
layers.Conv2D(64, (3, 3), activation='relu', name='Conv3'),
layers.MaxPooling2D((2, 2), name='MaxPool3'),
layers.Conv2D(64, (3, 3), activation='relu', name='Conv4'),
layers.MaxPooling2D((2, 2), name='MaxPool4'),
```

Figure 6 Code Snippet on defined Convolutional and Pooling Layers

The first convolutional layer applies 32 filters of size 3x3 to the input image, with the ReLU activation function being applied to introduce non-linearity. The first pooling layer, performs max pooling with a 2x2 window, reducing the spatial dimensions of the feature map by taking the maximum value in each 2x2 patch. As seen in Figure 6, the next layers for both convolutional and pooling follow the same pattern as Conv1 and MaxPool1, with the difference that it is defined with 64 filters of size 3x3. Each convolutional layer is followed by a max pooling layer, progressively reducing the spatial dimensions of the feature maps while increasing the depth.

This layer transforms the raw input images into a set of high-level features that are then fed into the fully connected layers for final classification. This structure allows the model to effectively learn and recognize the different conditions of the corn and maize plants.

c) Classification Layer: Fully Connected

The dense layer or fully connected layer takes the input from 1D vector. For Technique 3, the output for Texture Extraction is a kind of vector. Hence, it can be straight away inputted into the model because the input is compatible.

```
# Define the model architecture
n_classes = len(label_to_index)
input_shape = X_train.shape[1]

model = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(input_shape,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(n_classes, activation='softmax')
])
```

Figure 7 Code Snippet on defined Fully Connected Layer for Technique 3

Figure 7 shows the code snippet defining the whole architecture of Technique 3's classification model. The first layer has 128 neurons and uses the ReLU activation function, while the second layer has 64 neurons but also uses the ReLU activation function. It further processes the information from the previous layer. The final dense layer has n_classes neurons, which are the 4 classes. It uses the softmax activation function to output a probability distribution over the classes, allowing for multi-class classification of the maize condition.

For the other two techniques, namely Technique 1 and Technique 2, the output of the previous layer is 3 dimensional, it needs to be flattened before being fed into this layer. To do that, the first line of code in Figure 8 shows the library from TensorFlow is used.

```
layers.Flatten(name='Flatten'),
layers.Dense(64, activation='relu', name='Dense1'),
layers.Dense(n_classes, activation='softmax', name='Output')
```

Figure 8 Code Snippet on defined Fully Connected Layer for Technique 1 and 2

Once the output is fed, the layers are defined as in Figure 8, in the way that Dense1 is used. Dense1 is a fully connected layer with 64 units of neurons. It connects each neuron in this layer to every neuron in the previous layer. This layer consolidates the features extracted by the convolutional layers and prepares them for the final classification in classifying the corn condition as either, Blight, Gray Leaf Spot, Common Rust, and Healthy.

d) Output

The output of this model is a probability distribution over the four classes for each input image. The class with the highest probability is the predicted class for that image. The softmax activation function as defined in both types of architecture ensures that the output probabilities sum to 1, making them interpretable as probabilities.

9.0 Results of Training and Validation Dataset

a) Learning Rate

The learning rate is specified when compiling the model. For this model, the learning rate is set to the default value for both Adamax (Technique 1 and 2) and Adam (Technique 3) optimizers as shown in Figure 9. Specifically, for the Adamax optimizer, the learning rate is explicitly set to 0.001, ensuring consistent usage of this value throughout training.

```
# Compile the model
model.compile(optimizer=Adamax(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])
```

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

Figure 9 Code Snippets on defined Learning Rate

b) Training Model

For Techniques 1 and 2, data preprocessing includes augmentation techniques applied to enhance model robustness and performance. Data generators (train_gen for training, valid_gen for validation) dynamically load and preprocess batches, optimizing memory use and enabling real-time augmentation. For Technique 1, the model is built and trained by iterating through the batches of training data yielded by train_gen for 4 epochs, as seen in Figure 10. For each epoch, the model computes gradients, updates its internal parameters and weights based on the Adamax optimizer and categorical_crossentropy loss function, and evaluates its performance on the validation data yielded by valid_gen. For Technique 2, the model follows a similar process but is trained for 10 epochs. This longer training period allows the model to potentially capture more complex patterns in the data, further enhancing its performance.

```
# Train the model
history = model.fit(
    train_gen,
    epochs=4,
    batch_size=32,
    verbose=1,
    validation_data=valid_gen,
    callbacks=[callbacks]
)
```

Figure 10 Code Snippet on Model Training for Techniques 1 and 2

While for Technique 3, a different training approach is utilized. Figure 11 shows that the model is trained directly using the `X_train` and `y_train` arrays. Each epoch involves iterating over the entire dataset (`X_train` and `y_train`), computing gradients, updating model parameters, and evaluating performance on the validation data (`X_valid` and `y_valid`). The `callbacks` parameter allows for the incorporation of additional functionality during training, such as logging metrics or early stopping based on certain conditions.

```
# Train the model
history = model.fit(
    X_train, y_train,
    epochs=10,
    batch_size=32,
    verbose=1,
    validation_data=(X_valid, y_valid),
    callbacks=[callbacks]
)
```

Figure 11 Code Snippet on Model Training for Technique 3

The history object returned by model.fit contains information about the training process, such as loss and accuracy metrics recorded after each epoch. This information can be used to analyze how well the model is learning and to diagnose any issues during training, such as overfitting or underfitting.

c) Accuracy, Loss, and Error Rate

- **Technique 1 - AHE**

Table 6 shows the result of the training and validation dataset for the technique of AHE.

Table 6 Result of Training and Validation Dataset for Technique 1

Epochs	Training Accuracy	Validation Accuracy	Training Loss	Validation Loss	Training Error Rate	Validation Error Rate
1	0.6738	0.7707	5.9724	9.5390	0.3262	0.2293
2	0.8207	0.8089	0.4170	0.4425	0.1793	0.1911
3	0.8716	0.8280	0.3077	0.4123	0.1284	0.1720
4	0.9017	0.8280	0.2401	0.3919	0.0983	0.1720

The training accuracy improved steadily across the epochs, starting from 67.38% in the first epoch and reaching 90.17% by the fourth epoch. This consistent increase indicates that the model is effectively learning from the training data. In parallel, the validation accuracy also showed improvement, starting from 77.07% in the first epoch and stabilizing around 82.80% in the third and fourth epochs, suggesting that the model is generalizing well to the validation data.

Alongside accuracy, the training loss decreased significantly from 5.9724 in the first epoch to 0.2401 in the fourth epoch. A lower training loss over epochs signifies that the model's predictions are increasingly aligning with the actual values on the training data. Similarly, the validation loss saw a reduction from 9.5390 in the first epoch to 0.3919 by the fourth epoch, indicating that the model's predictions are improving in accuracy on unseen data.

In terms of error rates, the training error rate, calculated as 1 - training accuracy, decreased from 32.62% in the first epoch to 9.83% in the fourth epoch. This decline reflects an increasing proportion of correct predictions on the training data. The validation error rate, calculated as 1 - validation accuracy, also decreased from 22.93% in the first epoch to 17.20% in the fourth epoch, showing that the model is making fewer errors on the validation data as training progresses.

Overall, these metrics demonstrate the model's effective learning and generalization capabilities across the epochs. The improvements in both accuracy and loss for the training and validation datasets as seen in Figure 12 highlight the model's robustness and its ability to perform well on both seen and unseen data.

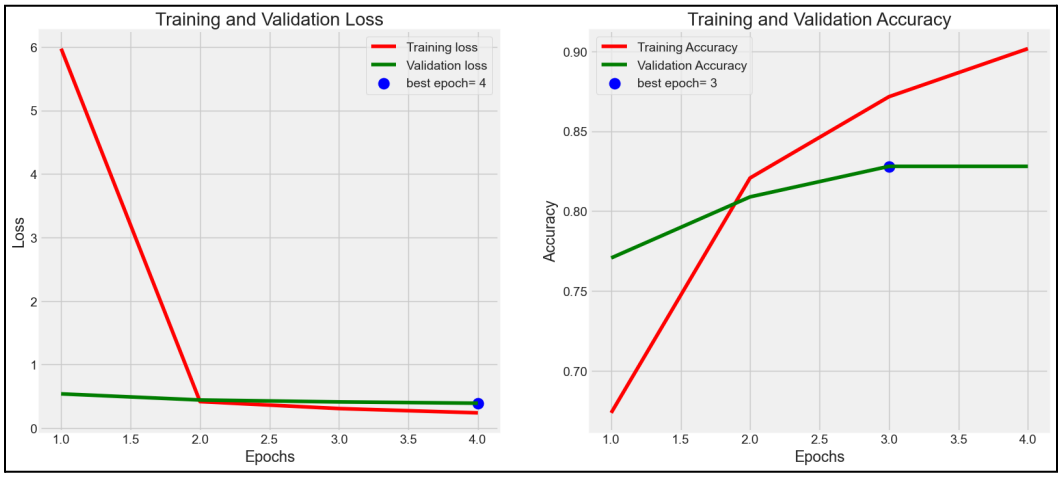


Figure 12 Result: Graph of Training and Validation Loss and Accuracy for Technique 1

- **Technique 2 - Canny Edge Detection**

Table 7 shows the result of the training and validation dataset for the technique of Canny Edge Detection.

Table 7 Result of Training and Validation Dataset for Technique 2

Epochs	Training Accuracy	Validation Accuracy	Training Loss	Validation Loss	Training Error Rate	Validation Error Rate
1	0.4227	0.6338	30.8632	0.8120	0.5773	0.3662
2	0.6708	0.7580	0.7125	0.6176	0.3292	0.2420
3	0.8305	0.8408	0.4110	0.4622	0.1695	0.1592
4	0.8754	0.8471	0.3282	0.4582	0.1246	0.1529
5	0.9219	0.8503	0.2092	0.4273	0.0781	0.1497
6	0.9527	0.8471	0.1387	0.4761	0.0473	0.1529
7	0.9735	0.8535	0.1045	0.4540	0.0265	0.1465
8	0.9788	0.8471	0.0866	0.5003	0.0212	0.1529
9	0.9855	0.8694	0.0609	0.4968	0.0145	0.1306
10	0.9938	0.8790	0.0385	0.4545	0.0062	0.1210

The training accuracy showed a significant improvement across the epochs, starting from 42.27% in the first epoch and reaching an impressive 99.38% by the tenth epoch. This consistent increase indicates that the model is effectively learning from the training data and refining its ability to make accurate predictions.

The validation accuracy also demonstrated a notable upward trend, beginning at 63.38% in the first epoch and achieving 87.90% by the tenth epoch. This improvement suggests that the model is not only performing well on the training data but is also generalizing effectively to the unseen validation data.

In terms of loss metrics, the training loss saw a dramatic decrease from 30.8632 in the first epoch to just 0.0385 by the tenth epoch. A lower training loss indicates that the model's predictions are becoming more accurate with each epoch. Similarly, the validation loss decreased from 0.8120 in the first epoch to 0.4545 by the tenth epoch, reflecting the model's enhanced performance on the validation dataset.

The training error rate decreased from 57.73% in the first epoch to a mere 0.62% by the tenth epoch. This reduction highlights the model's increasing precision in its predictions of the training data. Correspondingly, the validation error rate decreased from 36.62% in the first epoch to 12.10% by the tenth epoch, indicating a reduction in the model's prediction errors on the validation data.

Overall, these metrics collectively demonstrate the model's robust learning and generalization capabilities. The significant improvements in accuracy and reductions in loss and error rates across both training and validation datasets as seen in Figure 13 underscore the model's effectiveness in handling the given task.

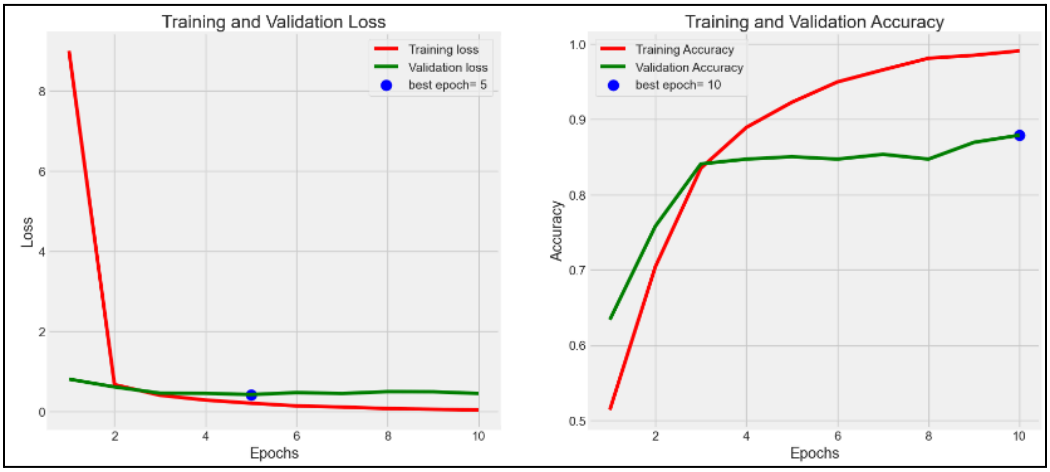


Figure 13 Result: Graph of Training and Validation Loss and Accuracy for Technique 2

- **Technique 3 - Texture Extraction**

Table 8 shows the result of the training and validation dataset for the technique of Texture Extraction.

Table 8 Result of Training and Validation Dataset for Technique 3

Epochs	Training Accuracy	Validation Accuracy	Training Loss	Validation Loss	Training Error Rate	Validation Error Rate
1	0.7272	0.7930	0.7213	0.4855	0.2728	0.2070
2	0.8280	0.8312	0.4471	0.4030	0.1720	0.1688
3	0.8427	0.8376	0.3940	0.4059	0.1573	0.1624
4	0.8488	0.8376	0.3707	0.3710	0.1512	0.1624
5	0.8536	0.8376	0.3552	0.3503	0.1464	0.1624
6	0.8573	0.8599	0.3494	0.3736	0.1427	0.1401
7	0.8629	0.8662	0.3401	0.3341	0.1371	0.1338
8	0.8688	0.8471	0.3259	0.3536	0.1312	0.1529
9	0.8640	0.8662	0.3248	0.3432	0.1360	0.1338
10	0.8685	0.8758	0.3210	0.3347	0.1315	0.1242

The training accuracy showed a consistent improvement over the 10 epochs, starting from 72.72% in the first epoch and reaching 86.85% by the tenth epoch. This trend suggests that the model effectively learned from the training data, enhancing its ability to correctly classify the training samples. Meanwhile, the validation accuracy increased from 79.30% in the first epoch to 87.58% by the tenth epoch, indicating that the model generalized well to the unseen validation data.

The training loss decreased significantly from 0.7213 in the first epoch to 0.3210 by the tenth epoch. This reduction implies that the model's predictions became more accurate as it learned from the training data.

Similarly, the validation loss showed a decline from 0.4855 in the first epoch to 0.3347 by the tenth epoch, reflecting an improvement in the model's performance on the validation dataset.

The training error rate dropped from 27.28% in the first epoch to 13.15% by the tenth epoch. This reduction demonstrates the model's increasing accuracy in predicting the correct labels for the training data. The validation error rate also decreased from 20.70% in the first epoch to 12.42% by the tenth epoch, indicating a reduction in the model's prediction errors on the validation data.

Overall, these metrics collectively demonstrate the model's robust learning and generalization capabilities. The significant improvements in accuracy and reductions in loss and error rates across both training and validation datasets, as seen in Figure 14 underscore the model's effectiveness in handling the given task.

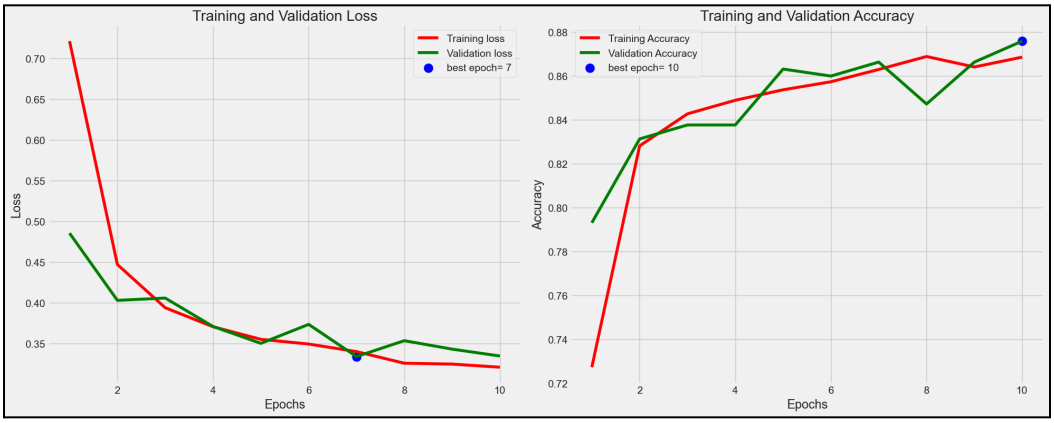


Figure 14 Result: Graph of Training and Validation Loss and Accuracy for Technique 3

10.0 Performance Evaluation

After the training and validation process, a CNN classification model that has been properly trained and validated is ready for use. This model can then be applied to make predictions on test data, as illustrated by the code in Figure 15. These predictions can be utilized to perform further performance evaluation, which helps assess the model's effectiveness in classifying corn leaf diseases. The subsequent sub-sections will delve into the classification report and confusion matrix for each technique. This analysis will identify the best model among the three and compare it with a model that does not incorporate image processing.

```
# Predictions on test data
preds = model.predict(test_gen)
y_pred = np.argmax(preds, axis=1)
```

Figure 15 Predictions on Test Data Code

a) Classification Report

This section evaluates three image processing techniques using their classification reports, which detail precision, recall, and F1 scores per class. These metrics provide a comprehensive view of each model's predictive capabilities, helping identify the best-performing technique.

- **Technique 1 - AHE**

Based on Figure 16, the results for AHE highlight strong performance in terms of overall accuracy (0.84) and weighted averages, particularly for the Common Rust and Healthy categories. These categories show high precision (0.91 and 0.95), recall (0.98 and 0.99), and F1-scores (0.95 and 0.97) respectively. However, the metrics for Blight and Gray Leaf Spot are comparatively lower, indicating potential areas for improvement. Blight shows a precision of 0.73, a recall of 0.71, and an F1-score of 0.72. The precision (0.60) and recall (0.49) for Gray Leaf Spot are the lowest among the categories, with an F1-score of 0.54, suggesting difficulties in accurately identifying this condition.

	precision	recall	f1-score	support
blight	0.73	0.71	0.72	86
common_rust	0.91	0.98	0.95	98
gray_leaf_spot	0.60	0.49	0.54	43
healthy	0.95	0.99	0.97	88
accuracy			0.84	315
macro avg	0.80	0.79	0.79	315
weighted avg	0.83	0.84	0.83	315

Figure 16 AHE Classification Report

- **Technique 2 - Canny Edge Detection**

Based on Figure 17, the results for the Canny Edge Detection technique demonstrate the same overall accuracy (0.84) as AHE, especially for the Common Rust and Healthy categories, with precision (0.91 and 0.97), recall (0.96 and 0.95), and F1-scores (0.94 and 0.96) respectively. The Blight category shows good precision (0.80) but slightly lower recall (0.73), resulting in an F1-score of 0.76. The Gray Leaf Spot category has the lowest performance metrics, with precision (0.54), recall (0.58), and an F1-score of 0.56, indicating challenges in accurately identifying this condition.

	precision	recall	f1-score	support
Blight	0.80	0.73	0.76	86
Common_Rust	0.91	0.96	0.94	98
Gray_Leaf_Spot	0.54	0.58	0.56	43
Healthy	0.97	0.95	0.96	88
accuracy			0.84	315
macro avg	0.80	0.81	0.81	315
weighted avg	0.85	0.84	0.84	315

Figure 17 Canny Edge Detection Classification Report

- **Technique 3 - Texture Extraction**

Based on Figure 18, the Texture Extraction technique achieves the highest overall accuracy (0.86) among the three techniques. It shows particularly strong results for the Common Rust and Healthy categories, with high precision (0.95 and 0.96), recall (0.97 and 0.93), and F1-scores (0.96 and 0.95) respectively. The Blight category exhibits a high recall (0.81) but slightly lower precision (0.75), resulting in an F1-score of 0.78. The Gray Leaf Spot category, while still the lowest, shows slight improvements in recall (0.58) compared to the other techniques, with precision (0.68) and an F1-score of 0.63. This technique provides a balanced performance across categories, making it a robust choice for texture analysis.

	precision	recall	f1-score	support
blight	0.75	0.81	0.78	86
common_rust	0.95	0.97	0.96	98
gray_leaf_spot	0.68	0.58	0.63	43
healthy	0.96	0.93	0.95	88
accuracy			0.86	315
macro avg	0.84	0.82	0.83	315
weighted avg	0.86	0.86	0.86	315

Figure 18 Texture Extraction Classification Report

- **Conclusion**

In conclusion, Techniques 1 (AHE) and 2 (Canny Edge Detection) both achieve the same overall accuracy of 0.84. However, Technique 2 demonstrates better performance in terms of precision, recall, and F1 scores for most categories, making it the preferable option between the two. Despite this, Technique 3 (Texture Extraction) stands out as the best model among the three, achieving the highest overall accuracy and consistently strong performance across all metrics.

b) Confusion Matrix

The confusion matrix for each technique utilized will be discussed in this section. In this project, the confusion matrix will be used to evaluate the image classification model's performance in predicting the health status of corn leaves. Each row in the matrix indicates the actual labels, while each column represents the predicted labels. This matrix contains four classes: "Blight", "Common_Rust", "Gray_Leaf_Spot", and "Healthy".

- **Technique 1 - AHE**

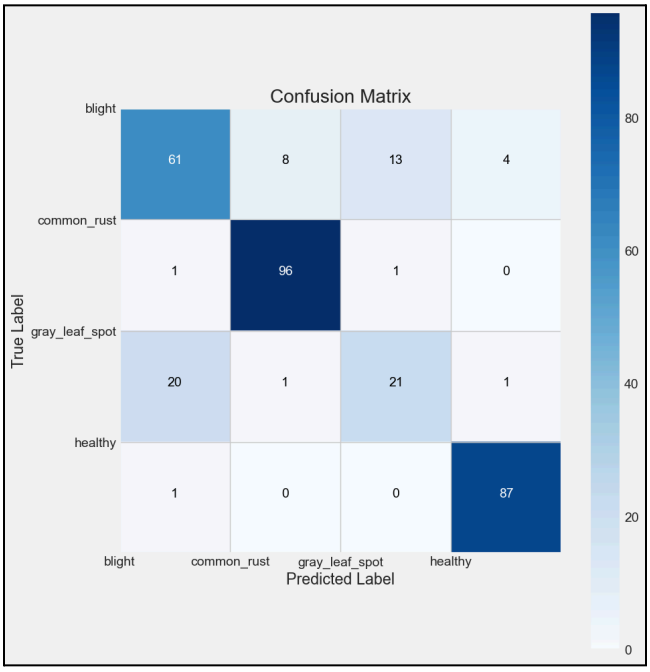


Figure 19 AHE Confusion Matrix

Based on Figure 19 above, shows the confusion matrix for the corn disease image classification model using the AHE technique. In this confusion matrix, the “Blight” class label correctly classified 61 images. But, on the other hand, there were some misclassifications for this class which was 8 images classified as “Common_Rust”, 13 as

“Gray_Leaf_Spot”, and 4 as “Healthy”. This means that while this model performs well in identifying the “Blight” class, there is still some confusion in classifying the “Blight” class, especially misclassifying it with the “Gray_Leaf_Spot” class.

Now, in the “Common_Rust” class label, this model shows excellent performance by correctly classifying 96 images. For this class label, the misclassifications were minimal which depicted 1 image incorrectly classified as “Blight”, and 1 as “Gray_Leaf_Spot”. This high accuracy indicates that this model is very effective in identifying “Common_Rust”.

However, the “Gray_Leaf_Spot” class label illustrates that there were some challenges faced by the model which was observed by its low accuracy. Only 21 images were correctly classified in this class label, while 20 were misclassified as “Blight”, 1 as “Common_Rust”, and 1 as “Healthy”. This significant level of misclassification, especially with “Blight”, highlighted the difficulties for this model in accurately identifying “Gray_Leaf_Spot”.

For the “Healthy” class label, the model shows strong performance by correctly classifying 87 images. In this class label, there was only 1 misclassification which was classified as the “Blight” class. This suggests that the model is reliable in distinguishing healthy leaves from those affected by diseases.

Overall, the confusion matrix suggests that the classification model using AHE performs well for “Blight”, “Common_Rust” and “Healthy” class labels, but not for “Gray_Leaf_Spot”.

- **Technique 2 - Canny Edge Detection**

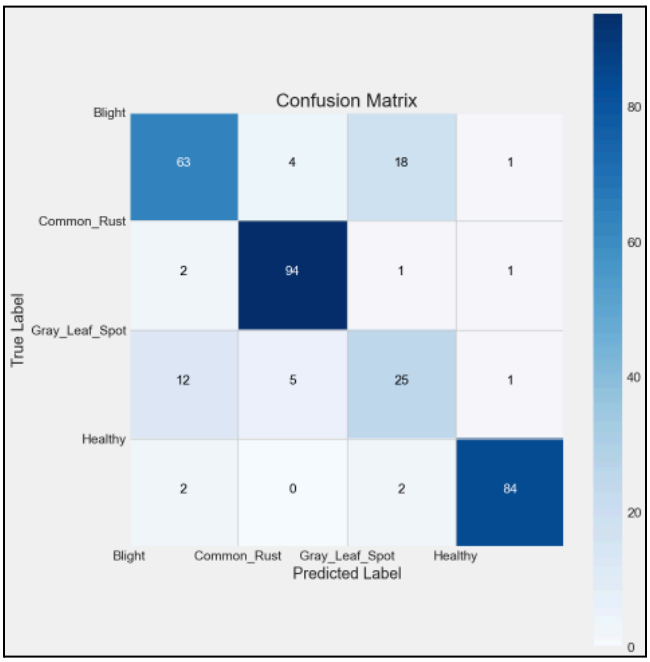


Figure 20 Canny Edge Detection Confusion Matrix

Based on Figure 20, for the “Blight” class label, the corn disease image classification model which incorporates the Canny Edge Detection technique correctly classified 63 images, indicating a strong performance in this class. However, there were some misclassifications in which 4 images were incorrectly classified as “Common_Rust”, 18 as “Gray_Leaf_Spot”, and 1 as “Healthy”. This suggests that while the model performs well, there is some confusion, especially with the “Gray_Leaf_Spot” class.

The “Common_Rust” class label shows impressive performance, with 94 images correctly classified. There were minimal misclassifications in this class when 2 images were incorrectly classified as “Blight”, 1 as “Gray_Leaf_Spot”, and 1 as “Healthy”. This indicates that the model has high accuracy in identifying “Common_Rust”.

The “Gray_Leaf_Spot” class label, however, shows the lowest accuracy. Only 25 images were correctly classified, while there were notable misclassifications into other classes. Specifically, 12 images were misclassified as “Blight”, 5 as “Common_Rust”, and 1 as “Healthy”. This indicates a significant challenge in accurately identifying “Gray_Leaf_Spot”, with a tendency to confuse it with other classes, particularly “Blight”.

For the “Healthy” class label, the model again shows strong performance, correctly classifying 84 images. There were minimal misclassifications as there were only 2 images that were incorrectly classified as “Blight”, and 2 as “Gray_Leaf_Spot”. This suggests that the model is quite reliable in distinguishing healthy leaves from those affected by diseases.

Overall, the confusion matrix suggests that the classification model using this Canny Edge Detector technique performs well for the “Common_Rust” and “Healthy” classes, but there is room for improvement in distinguishing between “Blight” and “Gray_Leaf_Spot”.

- **Technique 3 - Texture Extraction**

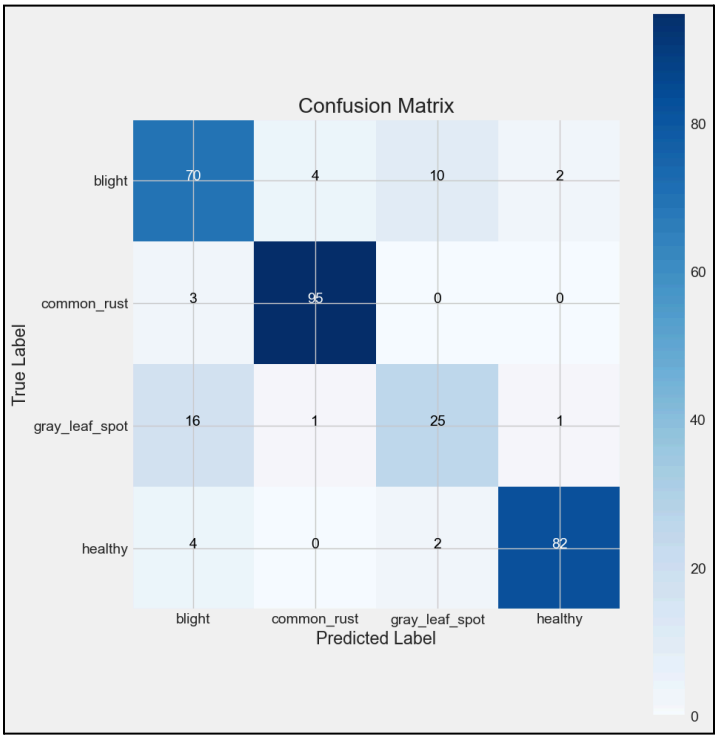


Figure 21 Texture Extraction Confusion Matrix

Figure 21 displays the confusion matrix for the corn disease image classification model using the texture extraction technique. In this model, the “Blight” class label had successfully classified 70 images correctly which depicts a good performance in this class. Although there were some misclassifications regarding this class label in which 4 images were incorrectly classified as “Common_Rust”, 10 as “Gray_Leaf_Spot”, and 2 as “Healthy”, this model still performs well in classifying the “Blight” class.

The “Common_Rust” class label shows the most impressive performance among the four class labels. This class label was predicted with 95 images correctly while only having 3 misclassifications that were confused with the “Blight” class. This means that this model has high accuracy in identifying the “Common_Rust” class label.

In this model, out of the four class labels, the “Gray_Leaf_Spot” class label is the one with the lowest accuracy. In this confusion matrix, there were only 25 correct image predictions. The second highest image predictions that were misclassified in this class label, which is “Blight”, have a high number of predictions at around 16 even though it is a misclassification. For the misclassification for this class label with “Common_Rust” and “Healthy” classes, both have 1 misclassification. This indicates that this model has a rough time in accurately identifying “Gray_Leaf_Spot”, as it tends to be confused with the “Blight” class.

For the “Healthy” class label, this model displays strong performance by correctly classifying 82 images. There were minimal misclassifications as there were only 4 images that were incorrectly classified as “Blight”, and 2 as “Gray_Leaf_Spot”. This means that this model in distinguishing healthy leaves from those affected by diseases can be quite reliable.

Hence, the confusion matrix suggests that the performance of corn disease classification models using texture extraction techniques is very good in distinguishing the “Blight”, “Common_Rust”, and “Healthy” classes, but it is still not quite reliable in predicting “Gray_Leaf_Spot” as it highly to mismatch with “Blight” class.

- **Conclusion**

In conclusion, the confusion matrices for Techniques 1 (AHE), 2 (Canny Edge Detection), and 3 (Texture Extraction) reveal distinct strengths and weaknesses in the models' performances. Techniques 1 and 2 show good overall performance but face challenges in accurately identifying "Gray_Leaf_Spot," often confusing it with "Blight." Technique 3, while also struggling with "Gray_Leaf_Spot," demonstrates the most robust performance, particularly excelling in the "Blight," "Common_Rust," and

"Healthy" class labels. Therefore, among the three, Technique 3 (Texture Extraction) proves to be the most effective, providing the best overall classification accuracy and reliability in distinguishing between the different disease classes and healthy leaves.

c) Best Model

- **CNN Model with Technique 3**

Based on the comparison of the three image classification models using different techniques which can be referred to in the classification report and confusion matrix sections earlier, the Model with technique 3 which is the Texture Extraction technique was deemed to be the best model. This is due to its having the highest accuracy score at 0.86 compared to the other two models which have the same accuracy score of 0.84. On top of that, out of the three models, the confusion matrix for the model with technique 3 shows the best performance in distinguishing and classifying the different classes of corn diseases, particularly the “Blight”, “Common_Rust”, and “Healthy” classes. However, it is still not very reliable in predicting the "Gray_Leaf_Spot" class because it has a significant amount of mismatch with the "Blight" class. In summary, the model using the Texture Extraction technique (Technique 3) outperforms the others, achieving the highest accuracy and demonstrating superior performance.

- **CNN Model without Image Processing**

For this CNN model without image processing for classifying corn diseases, these results, which are classification reports and confusion matrix, were obtained from this Kaggle notebook: CNN Model without Image Processing. Based on Figure 22, the model without employing an image processing technique achieves an accuracy score of 0.70. It displays quite strong results for the Common Rust and Healthy categories, with high precision (0.92 and 0.89), recall (0.85 and 0.82), and F1-scores (0.88 and 0.85) respectively. The Blight category has a quite low result with precision (0.61), recall (0.49), and F1-scores (0.54). Out of the four classes, the Gray Leaf Spot category shows the lowest performance, with a precision of 0.29, a recall of 0.51, and an F1-score of 0.37. This means that the model has the most hard time in classifying the Gray_Leaf_Spot class.

	precision	recall	f1-score	support
Blight	0.61	0.49	0.54	86
Common_Rust	0.92	0.85	0.88	98
Gray_Leaf_Spot	0.29	0.51	0.37	43
Healthy	0.89	0.82	0.85	88
accuracy			0.70	315
macro avg	0.68	0.67	0.66	315
weighted avg	0.74	0.70	0.71	315

Figure 22 Model Without Image Processing Technique Classification Report

The confusion matrix shown in Figure 23 illustrates the performance of the corn disease image classification model without applying any image processing techniques. Just like previous confusion matrices using image processing techniques, 4 class labels were predicted as the image data used are the same.

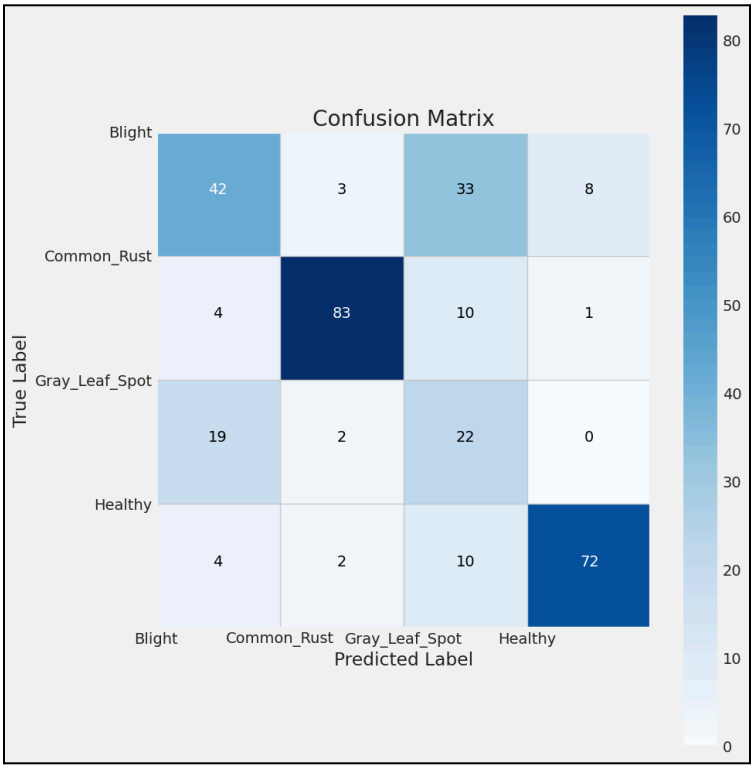


Figure 23 Model Without Image Processing Technique Confusion Matrix

In this model, the "Blight" class label was correctly classified with 42 images. However, it also misclassified 33 images as "Gray_Leaf_Spot," 8 as "Healthy," and 3 as "Common_Rust." These misclassifications show that although the model can detect "Blight" to a certain degree, there is still an extensive amount of confusion, especially when it comes to the "Gray_Leaf_Spot" class.

The "Common_Rust" class label for this model demonstrates strong performance, with 83 images correctly classified. There were minimal

misclassifications as there were only 15 misclassifications where 10 images were incorrectly classified as "Gray_Leaf_Spot," 4 as "Blight," and 1 as "Healthy." This high accuracy signifies the model's effectiveness in identifying "Common_Rust," despite a few confusion with other classes.

With only 22 images correctly classified, the "Gray_Leaf_Spot" class label, on the other hand, had the lowest accuracy. 19 images were incorrectly classified by the model as "Blight," 2 as "Common_Rust," and 0 as "Healthy." The substantial amount of misclassification, particularly for "Blight," emphasizes how difficult it is for the algorithm to recognize "Gray_Leaf_Spot."

The model shows great performance for the "Healthy" class label, properly classifying 72 images. 10 images were incorrectly classified as "Gray_Leaf_Spot," 4 as "Blight," and 2 as "Common_Rust." Misclassification rates in this class label were relatively low. This suggests that the model can accurately identify healthy leaves from unhealthy ones.

In summary, this confusion matrix indicates that the model does a good job of recognizing the "Common_Rust" and "Healthy" classes, but has trouble correctly classifying "Blight" and "Gray_Leaf_Spot." It is evident from the substantial amount of mismatch that exists between "Blight" and "Gray_Leaf_Spot" that these classes require better differentiation techniques.

- ## Conclusion

In conclusion, the model using the Texture Extraction technique stands out as the best for corn disease classification, achieving the highest accuracy (0.86) and demonstrating strong performance across most categories, especially Common Rust and Healthy, compared to the CNN model

without image processing, which has an accuracy of 0.70. To conclude, the use of image processing techniques like Texture Extraction significantly enhances the overall performance, providing better precision, recall, and F1-scores for the classification model. Therefore, it improves the model's reliability and effectiveness in distinguishing between different corn diseases.

11.0 Summary

In summary, this project aims to leverage advanced image processing techniques to enhance crop health monitoring, specifically targeting diseases affecting corn plants. The chosen dataset, "Corn or Maize Leaf Disease Dataset" from Kaggle, provides a comprehensive collection of images representing both healthy and diseased states of corn leaves, focusing on common rust, gray leaf spot, and blight. The dataset's extensive collection of 4,188 images across four classes ensures a robust foundation for developing a reliable classification model.

The project's primary objectives are to identify effective image processing techniques, develop a classification model, and evaluate the accuracy of the classification system in classifying corn leaf diseases. The methodology includes image enhancement techniques such as color correction and contrast enhancement to improve image quality. After image enhancement, three different techniques were performed: AHE, Canny Edge Detection, and Texture Feature Extraction. Each of these techniques was then used to train a CNN model. The performance of these three models was evaluated using confusion matrices and classification reports. Among these, the Texture Feature Extraction technique model was proven to be the best for classifying corn diseases.

Overall, the project demonstrates the significant impact of incorporating advanced image processing techniques in improving the accuracy and reliability of corn disease classification. By leveraging these techniques and evaluating

their performance, the project contributes to the development of more efficient tools for early disease detection, thereby promoting healthier crop production and sustainable farming practices.

In the future, the project aims to explore advanced feature enhancement techniques such as multi-modal fusion and adversarial feature learning to enhance the discriminative power of extracted features, thereby improving classification accuracy. Implementation of CNNs, specialized for image classification, will remain pivotal in categorizing corn plant images into disease categories and healthy states. By integrating CNNs with advanced image processing methods, the project aims to enhance early disease detection capabilities, facilitating timely interventions that promote sustainable farming practices, optimize resource utilization, and ultimately increase agricultural yields. These efforts underscore the project's commitment to leveraging innovative technologies to revolutionize crop health monitoring and support more resilient agricultural systems.

References

J, ARUN PANDIAN; GOPAL, GEETHARAMANI (2019), “Data for: Identification of Plant Leaf Diseases Using a 9-layer Deep Convolutional Neural Network”, Mendeley Data, V1, doi: 10.17632/tywbtsjrjv.1

MD.Ismiel Hossen Abir. (2024, June 25). Corn or maize leaf disease. Kaggle. https://www.kaggle.com/code/mdismielhossenabir/corn-or-maize-leaf-disease

Omia, E., Bae, H., Park, E., Kim, M., Baek, I., Kabenge, I., … & Cho, B. (2023). Remote sensing in field crop monitoring: a comprehensive review of sensor systems, data analyses and recent advances. Remote Sensing, 15(2), 354. https://doi.org/10.3390/rs15020354

Shafi, U., Mumtaz, R., García-Nieto, J., Hassan, S., Zaidi, S., & Iqbal, N. (2019). Precision agriculture techniques and practices: from considerations to applications. Sensors, 19(17), 3796. https://doi.org/10.3390/s19173796\

Singh, D., Jain, N., Jain, P., Kayal, P., Kumawat, S., & Batra, N. (2019). PlantDoc: A Dataset for Visual Plant Disease Detection. arXiv. https://doi.org/10.48550/ARXIV.1911.10317

Smaranjit Ghose. (2020). Corn or Maize Leaf Disease Dataset. Retrieved from https://www.kaggle.com/datasets/smaranjitghose/corn-or-maize-leaf-disease-dataset/data.

Appendices

Appendix A

- **Import Packages**

```
# import packages
import os
import time
import shutil
import pathlib
import itertools
from PIL import Image, ImageEnhance

import cv2
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_style('darkgrid')
import matplotlib.pyplot as plt

from skimage import feature, exposure
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam, Adamax
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Activation, Dropout, BatchNormalization
from tensorflow.keras import regularizers
from tensorflow.keras import models, layers

import warnings
warnings.filterwarnings("ignore")

print('modules loaded')
```

- **Load Data Collection**

```
data_path = r"C:\Users\USER\OneDrive - Universiti Teknologi MARA\Do

blight = []
common_rust = []
gray_leaf_spot = []
healthy = []

for subfolder in os.listdir(data_path):
    subfolder_path = os.path.join(data_path, subfolder)
    if not os.path.isdir(subfolder_path):
        continue

    for image_filename in os.listdir(subfolder_path):
        image_path = os.path.join(subfolder_path, image_filename)

        if subfolder == 'Blight':
            blight.append(image_path)
        elif subfolder == 'Common_Rust':
            common_rust.append(image_path)
        elif subfolder == 'Gray_Leaf_Spot':
            gray_leaf_spot.append(image_path)
        elif subfolder == 'Healthy':
            healthy.append(image_path)

# Create a dictionary to store all arrays
image_data = {
    'blight': blight,
    'common_rust': common_rust,
    'gray_leaf_spot': gray_leaf_spot,
    'healthy': healthy
}

# Create a DataFrame for overview
data = pd.DataFrame({
    'blight': pd.Series(blight),
    'common_rust': pd.Series(common_rust),
    'gray_leaf_spot': pd.Series(gray_leaf_spot),
    'healthy': pd.Series(healthy)
})
```

Appendix B

- **Image Processing**
  - **Adaptive Histogram Equalization (Technique 1)**

```python
# Adaptive Histogram Equalization - Technique 1
def image_processing(image_path, label, index):
    # Load image
    ori_image = cv2.imread(image_path)

    # Perform image enhancement first
    resized_image = image_enhancement(ori_image)

    # Enhance contrast using CLAHE (Adaptive Histogram Equalization)
    image_enhanced = exposure.equalize_adapthist(resized_image)

    # Create folder path for saving processed images
    save_folder = os.path.join(processed_data_path, label_to_folder[label])
    os.makedirs(save_folder, exist_ok=True)

    # Create file name based on label and index
    file_name = f"Corn_{label.capitalize()}_{index + 1}.jpg"
    save_path = os.path.join(save_folder, file_name)

    # Save the processed image
    cv2.imwrite(save_path, (image_enhanced * 255).astype(int))

    return save_path

# Iterate through each image path, preprocess and save
for index, row in data.iterrows():
    print(f"Processing index {index}/{len(data) - 1}...")

    for label in label_to_folder:
        image_path = row[label]

        try:
            save_path = image_processing(image_path, label, index)
        except Exception as e:
            print(f"Image path for label '{label}': {image_path}")
            print(f"Error processing image: {e}")
            continue

    if index == 0:
        processed_image = cv2.imread(save_path)
        plt.figure(figsize=(10, 10))
        plt.imshow(processed_image)
        plt.axis("off")
        plt.show()

print("Image preprocessing and saving complete.")
```

- **Canny Edge Detection (Technique 2)**

```python
def canny_edge_detector(enhanced_image):
    # Perform Canny Edge Detector
    edged = cv2.Canny(enhanced_image, threshold1=50, threshold2=150)

    return edged
```

```python
def apply_mask(image, edge):
    # Apply the mask to the enhanced image
    masked_image = cv2.bitwise_and(image, image, mask=edge)
    return masked_image
```

```python
procImg = []
```

```python
# Process all images and show one example
for index, row in data.iterrows():
    # Original image
    original_image = row['image']

    # Enhanced image
    corrected_image, final_image, enhanced_image = image_enhancement(original_image)

    # Canny Edged image
    canny_edged_image = canny_edge_detector(enhanced_image)

    # Masked image
    applied_image = apply_mask(enhanced_image, canny_edged_image)

    # Append masked image to list
    procImg.append(applied_image)
```

```python
# Assuming procImg is your list of processed images from the previous part
for index in range(len(procImg)):
    image = procImg[index]
    imageRGB = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Define the base directory to save processed images
    base_output_dir = r"C:/Users/User/Documents/CS230-SciComp/Sem6/CSC566/MINI PROJECT/edged_images"

    # Create the base output directory if it doesn't exist
    os.makedirs(base_output_dir, exist_ok=True)

    label = data.loc[index, 'label']
    label_dir = os.path.join(base_output_dir, label)

    # Create subdirectory for the label if it doesn't exist
    os.makedirs(label_dir, exist_ok=True)

    # Generate filename based on index
    filename = f"leaf_{index}.jpg"
    filepath = os.path.join(label_dir, filename)

    # Save the leaf image
    cv2.imwrite(filepath, imageRGB)

print("Leaf masked image saving process completed.")
```

- **Texture Extraction (Technique 3)**

```python
# Texture Extraction - Technique 3
def image_processing(image_path):
    # Load and preprocess image
    ori_image = cv2.imread(image_path)

    # Perform image enhancement first
    resized_image = image_enhancement(ori_image)

    # Extract texture features using Local Binary Pattern (LBP)
    texture_features = extract_texture_features(resized_image)

    return ori_image, texture_features

# Function focus extract the texture using LBP
def extract_texture_features(image):
    # Convert image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    # Compute Local Binary Pattern (LBP) features
    lbp_radius = 3
    lbp_points = 24
    lbp = feature.local_binary_pattern(gray_image, lbp_points, lbp_radius, method='uniform')

    # Calculate histogram of LBP and normalize it
    hist_lbp, _ = np.histogram(lbp.ravel(), bins=np.arange(0, lbp_points + 3), range=(0, lbp_points + 2))
    hist_lbp = hist_lbp.astype("float")
    hist_lbp /= (hist_lbp.sum() + 1e-7)

    return hist_lbp
```

```python
# List to store image paths and corresponding texture features
processed_data = []

# Iterate over each image path and extract features
for label, paths in image_data.items():
    print(f"Processing {label} images...")
    for idx, image_path in enumerate(paths):
        print(f"\rProcessing image {idx + 1}/{len(paths)}", end='')
        original_image, texture_features = image_processing(image_path)
        processed_data.append({'image_path': image_path, 'label': label, 'texture_features': texture_features})
    print()  # To move to the next line after completing processing of label images

# Convert processed_data to DataFrame for easier manipulation
processed_df = pd.DataFrame(processed_data)

# Save DataFrame to CSV for later use
processed_df.to_csv(r'C:\Users\USER\OneDrive - Universiti Teknologi MARA\Documents\STUDY\SEM 6\CSC566\dataset\p

print("Processing complete. CSV file saved.")
```

Appendix C

- **Image Enhancement**

```python
# Function to perform color correction
def image_enhancement(image_path):
    # Load the image
    image = cv2.imread(image_path)
    # Convert color if necessary
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Color Correction
    lab_image = cv2.cvtColor(image, cv2.COLOR_RGB2LAB)
    l_channel, a_channel, b_channel = cv2.split(lab_image)
    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8, 8))
    cl = clahe.apply(l_channel)
    limg = cv2.merge((cl, a_channel, b_channel))
    corrected_image = cv2.cvtColor(limg, cv2.COLOR_LAB2RGB)

    # Contrast enhancement
    pil_image = Image.fromarray(corrected_image)
    enhancer = ImageEnhance.Contrast(pil_image)
    enhanced_image = enhancer.enhance(1.5)  # Enhance contrast by a factor of 1.5

    # Convert back to numpy array (RGB)
    final_image = np.array(enhanced_image)

    # Pad the image to make it square
    final_image = pad_to_square(final_image)

    # Resize image to target size
    img_size = (256, 256)
    resized_image = cv2.resize(final_image, img_size)
    resized_image = cv2.cvtColor(resized_image, cv2.COLOR_RGB2BGR)

    return resized_image

# Function to add padding to the image to make it square
def pad_to_square(image):
    height, width, _ = image.shape
    max_dim = max(height, width)
    top = (max_dim - height) // 2
    bottom = max_dim - height - top
    left = (max_dim - width) // 2
    right = max_dim - width - left
    padded_image = cv2.copyMakeBorder(image, top, bottom, left, right, cv2.BORDER_CONSTANT, value=[0, 0, 0])
    return padded_image
```

Appendix D

- **Data Splitting**
  - **Techniques 1 and 2**

```
# Base directory where processed images are saved
base_output_dir = r"C:\Users\USER\OneDrive - Universiti Teknologi MARA\Documents\STUDY\SEM 6\CSC566\dataset\pr

# Initialize lists to store image paths and labels
images = []
labels = []

# Iterate through each label folder in the base output directory
for label in os.listdir(base_output_dir):
    label_dir = os.path.join(base_output_dir, label)

    if os.path.isdir(label_dir):
        # Iterate through each image file in the label folder
        for filename in os.listdir(label_dir):
            if filename.endswith('.jpg'):
                # Construct the image path
                image_path = os.path.join(label_dir, filename)
                images.append(image_path)
                labels.append(label)

# Create a DataFrame from the collected image paths and labels
data = pd.DataFrame({'image': images, 'label': labels})

# Split the dataset into train (85%), validation (7.5%), and test (7.5%) sets
strat = data['label']
train_df, temp_df = train_test_split(data, train_size=0.85, shuffle=True, random_state=123, stratify=strat)
strat = temp_df['label']
valid_df, test_df = train_test_split(temp_df, train_size=0.5, shuffle=True, random_state=123, stratify=strat)
```

```
# Optionally, reset the index for each dataframe
train_df = train_df.reset_index(drop=True)
valid_df = valid_df.reset_index(drop=True)
test_df = test_df.reset_index(drop=True)

# Display the sizes of each dataset (optional)
print(f"Number of samples in train set: {len(train_df)}")
print(f"Number of samples in validation set: {len(valid_df)}")
print(f"Number of samples in test set: {len(test_df)}")

# Define image generator parameters
batch_size = 32
img_size = (256, 256)
channels = 3
img_shape = (img_size[0], img_size[1], channels)

# Initialize ImageDataGenerators
tr_gen = ImageDataGenerator()
ts_gen = ImageDataGenerator()

# Create data generators
train_gen = tr_gen.flow_from_dataframe(train_df, x_col='image', y_col='label', target_size=img_size,
                                       class_mode='categorical', color_mode='rgb', shuffle=True,
                                       batch_size=batch_size)

valid_gen = ts_gen.flow_from_dataframe(valid_df, x_col='image', y_col='label', target_size=img_size,
                                       class_mode='categorical', color_mode='rgb', shuffle=True,
                                       batch_size=batch_size)

test_gen = ts_gen.flow_from_dataframe(test_df, x_col='image', y_col='label', target_size=img_size,
                                      class_mode='categorical', color_mode='rgb', shuffle=False,
                                      batch_size=batch_size)
```

- **Technique 3**

```
# Load processed data from CSV
processed_df = pd.read_csv(r'C:\Users\USER\OneDrive - Universiti Teknologi MARA\Documents\STUDY\SEM 6\CSC566\dataset\proces

# Convert texture features column from string to list
processed_df['texture_features'] = processed_df['texture_features'].apply(lambda x: np.fromstring(x.strip('[]'), sep=' '))

# Prepare feature and label arrays
X = np.vstack(processed_df['texture_features'].values)
label_to_index = {label: idx for idx, label in enumerate(processed_df['label'].unique())}
y = np.array([label_to_index[label] for label in processed_df['label']])

# Normalize features
X = (X - X.mean(axis=0)) / X.std(axis=0)

print("Label to Index Mapping:")
for label, index in label_to_index.items():
    print(f"{label}: {index}")
```

```
# Split the dataset into train (85%), validation (7.5%), and test (7.5%) sets
X_train, X_temp, y_train, y_temp = train_test_split(X, y, train_size=0.85, shuffle=True, random_state=123, stratify=y)
X_valid, X_test, y_valid, y_test = train_test_split(X_temp, y_temp, train_size=0.5, shuffle=True, random_state=123,
                                                    stratify=y_temp)

# Display the sizes of each dataset
print(f"Number of samples in train set: {len(X_train)}")
print(f"Number of samples in validation set: {len(X_valid)}")
print(f"Number of samples in test set: {len(X_test)}")
```