# INTEGRATION OF IMAGE PROCESSING ALGORITHM AND DEEP LEARNING APPROACHES TO MONITOR GINGER PLANT

**TAN CHENG YONG**

**A project report submitted in partial fulfilment of the requirements for the award of the degree of Bachelor of Engineering (Honours) in Electronic Systems**

**Faculty of Engineering and Green Technology**
**Universiti Tunku Abdul Rahman**

**January 2024**

# DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : _________________________

Name : TAN CHENG YONG

ID No. : 20AGB04834

Date :

# APPROVAL FOR SUBMISSION

I certify that this project report entitled **"INTEGRATION OF IMAGE PROCESSING ALGORITHM AND DEEP LEARNING APPROACHES TO MONITOR GINGER PLANT"** was prepared by **TAN CHENG YONG** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering (Honours) in Electronic Systems at Universiti Tunku Abdul Rahman.

Approved by,

Signature : _________________________

Supervisor : Dr. Lee Han Kee

Date : _________________________

The copyright of this report belongs to the author under the terms of the copyright Act 1987 as qualified by Intellectual Property Policy of Universiti Tunku Abdul Rahman. Due acknowledgement shall always be made of the use of any material contained in, or derived from, this report.

© 2024, Tan Cheng Yong. All right reserved.

Specially dedicated to

my beloved grandmother, mother and father

# INTEGRATION OF IMAGE PROCESSING ALGORITHM AND DEEP LEARNING APPROACHES TO MONITOR GINGER PLANT

## ABSTRACT

This study aims to integrate image processing and deep learning algorithms to monitor the growth of ginger plants. The proposed system is designed to detect ginger plants and track their growth rate effectively. The deep learning algorithm will undergo training using a dataset containing ginger plant images, which will allow it to accurately identify and categorize various stages of growth. The image processing techniques will be used to pre-process and enhance the quality of the images to making it easier for the deep learning model to identify the ginger plants. One YOLOv8 based model was developed for detecting and segmenting ginger plants in various growth states. Following the successful detection and segmentation of the plants, another YOLOv8 based model was further developed to segment individual leaves from detected plant. In order to improve the monitoring process, a depth estimation model was used to calculate the distance from the camera to the plants, enabling measurements of the height and leaf area of the ginger plants. The integration of these two methods will provide a more efficient and reliable way to monitor ginger plant growth, which is important for farmers and researchers in the field of agriculture.

# TABLE OF CONTENTS

**DECLARATION** **ii**
**APPROVAL FOR SUBMISSION** **iii**
**ABSTRACT** **vi**
**TABLE OF CONTENTS** **vii**
**LIST OF TABLES** **x**
**LIST OF FIGURES** **xi**
**LIST OF SYMBOLS / ABBREVIATIONS** **xiii**
**LIST OF APPENDICES** **xiv**

**CHAPTER**

**1 INTRODUCTION** **1**
1.1 Background 1
1.2 Problem Statements 2
1.3 Aims and Objectives 3

**2 LITERATURE REVIEW** **4**
2.1 Deep Learning 4
2.2 Object Detection with Convolution Neural Network (CNN) 5
2.3 YOLO (You Only Look Once) 6
2.3.1 YOLO's Introduction 6
2.3.2 Evolution of YOLO Versions 6
2.3.3 YOLOv8 Object Detection Mechanism 8
2.3.4 YOLOv8 Architecture and Segmentation Mechanism 8
2.3.5 Advantages of YOLOv8 Segmentation 9

2.4 Faster R-CNN 10
2.5 Image enhancement 10
2.6 Leaf Segmenting 12
2.7 Leaf extraction and classification 13

**3 METHODOLOGY 15**
3.1 Introduction 15
3.2 Planning 16
3.2.1 Hardware Required 16
3.2.2 Software Required 16
3.2.3 Project Timeline & Resource allocation 17
3.3 Training 19
3.3.1 Dataset Preparation for YOLOv8 model training 19
3.3.2 Dataset Preparation for Depth Estimation Model Training 21
3.4 Implementation 22
3.4.1 Inference on New Images 22
3.4.2 Model Implementation 23
3.5 Analysis 24
3.5.1 Performance Analysis 24
3.6 Cost Estimation 26

**4 RESULTS AND DISCUSSIONS 27**
4.1 System Interface Results 27
4.2 Ginger Plant Detection Using YOLOv8 Model 29
4.2.1 Example of Ginger Plant Detection Results 29
4.2.2 Ginger Plant Detection Using YOLOv8 Model Performance 31
4.3 Leaf Detection and Health Classification Using YOLOv8 35
4.3.1 Example of Leaf Detection and Health Classification Results 35
4.3.2 Leaf Detection and Health Classification Using YOLOv8 Performance 37

4.4 Leaf Health Status Classification 41

4.4.1 Leaf Count Per Plant and Health Status Classification 42

4.5 Depth Estimation Model 43

4.5.1 Distance Measurement Results 43

4.5.2 Model Performance 45

4.6 Plant Height and Leaf Area Calculations 47

4.6.1 Example Calculation of Plant Height and Leaf Area 49

4.7 Challenges and Limitation 51

**5 CONCLUSION AND RECOMMENDATIONS 52**

5.1 Conclusion 52

5.2 Recommendation 53

**REFERENCES 54**

**APPENDICES 57**

# LIST OF TABLES

| TABLE | TITLE | PAGE |
| --- | --- | --- |
| Table 3.1: Gantt Chart for FYP1 | | 18 |
| Table 3.2: Gantt Chart for FYP2 | | 18 |
| Table 3.3: Hyperparameters use in Training Phase | | 19 |
| Table 3.4: Cost Estimation of Project Materials | | 26 |
| Table 4.1: Example Testing Results of YOLOv8 Segmentation Ginger Plant Detection Model | | 30 |
| Table 4.2: Example Testing Results of YOLOv8 Leaves Detection Model | | 36 |
| Table 4.3: Example Images of Detected and Classified Leaves on a Ginger Plant. | | 41 |
| Table 4.4: Leaf Health Status Confusion matrix | | 41 |
| Table 4.5: Leaf Health Status Confusion matrix | | 41 |
| Table 4.5: Example Depth Map of a Ginger Plant | | 43 |
| Table 4.6: Depth Estimation Result of Test Images | | 44 |
| Table 4.7: Comparison Between Calculated and Actual Plant Height | | 49 |

# LIST OF FIGURES


| FIGURE | TITLE | PAGE |
|---|---|---|
| Figure 2.1: Use of CNN in Object Detection | | 5 |
| Figure 2.2: Process of Image Enhancement | | 11 |
| Figure 3.1: Process Flow of Detecting the Ginger Plant | | 15 |
| Figure 3.2: Dataset of Plant Detection Model | | 20 |
| Figure 3.3: Dataset of Leaf Detection Model | | 21 |
| Figure 3.4: Overview of Model Implementation Phase | | 23 |
| Figure 4.1: Real-Time Plant Monitoring System Interface | | 27 |
| Figure 4.2: Real-Time Plant Monitoring System Interface | | 28 |
| Figure 4.3: Real-Time Plant Monitoring System Interface | | 28 |
| Figure 4.4: Labels in Training Process of Plants Detection | | 29 |
| Figure 4.5: F1-Confidence Curve | | 31 |
| Figure 4.6: Precision-Confidence Curve | | 32 |
| Figure 4.7: Precision-Recall Curve | | 33 |
| Figure 4.8 Recall-Confidence Curve | | 34 |
| Figure 4.9: Labels in Training Process of Leaves Detection | | 35 |
| Figure 4.10: F1-Confidence Curve | | 37 |
| Figure 4.11: Precision-Confidence Curve | | 38 |
| Figure 4.12: Precision-Recall Curve | | 39 |
| Figure 4.13: Recall-Confidence Curve | | 40 |

Figure 4.14: Graph of Predicted Distance vs. Actual Distance 44

Figure 4.15: Graph of Percentage Deviation vs. Actual Distance 45

Figure 4.16: Graph of Calculated Height and Actual Height vs. Actual Distance 50

Figure 4.17: Graph of Percentage Deviation vs. Actual Distance 50

## LIST OF SYMBOLS / ABBREVIATIONS

| | |
|---|---|
| $s_i$ | output of sigmod for the network |
| $s_k$ | skewness |
| | |
| $\alpha$ | weight hyperparameters |
| $\beta$ | weight hyperparameters |
| $E$ | Expected value |
| $\mu$ | mean value |
| $\sigma$ | standard deviation |
| | |
| ANNs | Artificial Neural Networks |
| BCE | Binary Cross Entropy |
| CMYK | Cyan, Magenta, Yellow, and Key (Black) |
| CNN | Convolutional Neural Network |
| CIOU | Complete Intersection over Union |
| DFL | Distribution Focal Loss |
| IoV | Internet of Vehicles |
| K-NN | k-nearest neighbours |
| LBP | Local Binary Patterns |
| R-CNN | Region-based Convolutional Neural Network |
| RGB | Red, Green, Blue |
| SPPF | Spatial Pyramid Pooling Fast |
| SSD | MobileNet Single Shot Detector |
| SVM | Support Vector Machines |
| TBoF | Trainable Bag of Freebies |
| YOLO | You Only Look Once |

## LIST OF APPENDICES


| APPENDIX | TITLE | PAGE |
| --- | --- | --- |
| A | Code for Training Yolo Model in Google Colab | 57 |
| B | Code for Real-Time Monitoring Interface in Python Language | 59 |
| C | Code for Target Detection in Python Language | 92 |
| D | Test Image used in Evaluated Depth Estimation Model | 100 |

# CHAPTER 1

# INTRODUCTION

## 1.1 Background

Agriculture is one of the leading industries in Malaysia and plays an important role in social and economic development. Malaysia has approximately 4.06 million hectares of agricultural land, 80 % of which is used for industrial corps such as rubber, palm oil, cocoa, coconut and pepper and some allocated for agriculture production (FONG, 1990). In 2009, the agriculture sector contributed RM20 bullion, or 4% of Malaysia's gross national income (GNI). In line with this, for a country like Malaysia, the need for economic growth in the agricultural sector has been growing at an alarming rate over the past few decades. As a result, the rate of production in the agriculture sector has doubled in the last two decades (Matahir & Tuyon, 2013).

For the export market, farmers in Malaysia manufacture a wide range of crop and grain goods. The ginger crop is one of these products that brings in an important amount of foreign funds for the country. Since the competence of agricultural extension workers and visual inspection are the main factors in traditional disease detection, it is costly and challenging to scale up early disease identification and classification, especially for mass production (Selvaraj, et al., 2019). In Malaysia with limited human and logistical infrastructure, smallholder farmers are less successful in addressing farming issues since they rely on their prior knowledge. Therefore, early detection of field diseases and growth rates is one of the important steps for early intervention to reduce the impact of food supply chains.

In recent years, the integration of image processing algorithms and deep learning techniques has shown great effect in addressing these challenges. By using these algorithms and techniques, visual data can be analysed with high accuracy, allowing for the automated detection and analysis of various plant characteristics. One of the deep learning algorithms, such as convolutional neural networks (CNNs) can be used to reducing the need for manual inspection and increasing the efficiency of the monitoring system. Among the cutting-edge technologies in this domain, YOLO (You Only Look Once) stands out as a particularly effective deep learning model for real-time object detection and segmentation (Wang, et al., 2024). Deep learning models can be trained on large datasets of ginger plant images, allowing them to learn the features and patterns associated with different growth stages. This help improve the accuracy of the monitoring system and provide more reliable information about the growth rate and health of the ginger plants. Additionally, the integration of image processing and deep learning approaches can also help address the challenges of variability in ginger plant appearance due to environmental factors such as soil conditions, sunlight exposure, and water availability.

## 1.2 Problem Statements

Monitoring of ginger plant growth and health is important in agriculture for optimizing yield and ensuring crop quality. However, traditional methods of plant monitoring depend on manual observation and measurement, which is labour-intensive, inconsistencies and likely to have human error. As the demand for precise agricultural practices increase, any delay or inaccuracy in detecting plant diseases or growth deficiencies can also lead to reduced yields and financial loss for farmers. Therefore, automated solutions that can provide accurate, real-time data on plant characteristics, such as height, leaf area, and overall health is required.

The lack of automated tools for efficient plant monitoring is a challenge, especially in large farms where manually plant assessment is inefficient. Furthermore, to determine the plant health and development, a more precise measurement technique is needed to assess plant height and leaf area. The integration of modern technologies

such as image processing and deep learning can provide solutions, but yet there is still lacking effective system specifically tailored to ginger plants.

This project aims to solve these issues by developing a system that integrates image processing algorithms with deep learning models to automate the detection, classification, and monitoring of ginger plants and their leaves. The system will not only detect and segment plants but also assess plant health and estimate plant height using depth estimation models, providing an efficient and accurate method for monitoring ginger crops.

## 1.3 Aims and Objectives

The objectives of the thesis are shown as following:

1. To design a model capable of detecting the ginger plants.
2. To design a model capable of detecting ginger leaf and classifying ginger leaf by its health status.
3. To design a model that can estimate the depth of the target to calculate its height and area.

# CHAPTER 2

# LITERATURE REVIEW

## 2.1 Deep Learning

Deep learning is being used more and more in monitoring plant growth as it has shown good performance in image classification. Deep learning is a type of machine learning that includes multi-layer artificial neural networks (ANNs) with multiple layers (Shrestha & Mahmood, 2019), where the model's outcomes and parameters are influenced by the examples used during training. Deep learning uses different types of learning methods include supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. Deep learning models have performed better than traditional machine learning models such as SVMs, k-NNs, and decision trees when it comes to monitoring plant growth, especially in the area of image-based plant phenotyping research, where deep learning models have been shown to be more effective than traditional machine learning models (Tong, et al., 2022).

Moreover, using deep learning models in plant growth monitoring can enhance the accuracy and efficiency of plant growth monitoring technologies, especially in the field of precision agriculture. It helps farmers to accurately predict crop yields, identify plant disease and determine the health of their plants, which enables them to make informed decisions regarding crop management and disease control. Development models created through deep learning can help researchers in gaining a clearer understanding of the factors influencing plant growth. This could also assist the researcher in creating a more efficient method for plant breeding and crop management.

Among deep learning networks, convolutional neural networks are more effective at capturing hierarchical patterns in image and video data due to the use of shared weights in convolution kernels. Convolutional neural networks are already used in agriculture for a variety of tasks, such as identifying diseases, classifying land cover, counting fruits, and identifying weeds through image analysis (Tong, et al., 2022). As of now, there have been 23 studies focused on deep learning applications for monitoring plant growth, released from 2017 to 2021, with most of them coming out in 2020 (Tong, et al., 2022). These researches show that the use of deep learning in monitoring plant growth is a new and developing area.

## 2.2 Object Detection with Convolution Neural Network (CNN)

CNN is a form of feed-forward neural network that uses weight sharing which is commonly used in object detection tasks. Convolution is a mathematical operation that demonstrates the overlapping of two functions by multiplying them together. The CNN architecture for object detection includes convolving the image with an activation function to produce feature maps, which are further processed with pooling layers to simplify spatial complexity and form abstracted feature maps. Furthermore, the feature maps are operated on by fully connected layers to produce an image recognition output, indicating the certainty of the predicted class labels (Pathak, et al., 2018). The layered architecture of CNN for object detection is shown in Figure 2.1. The CNN uses different types of pooling layers to enhance efficiency and decrease parameters, these layers are translation-invariant and process each patch within the chosen map.

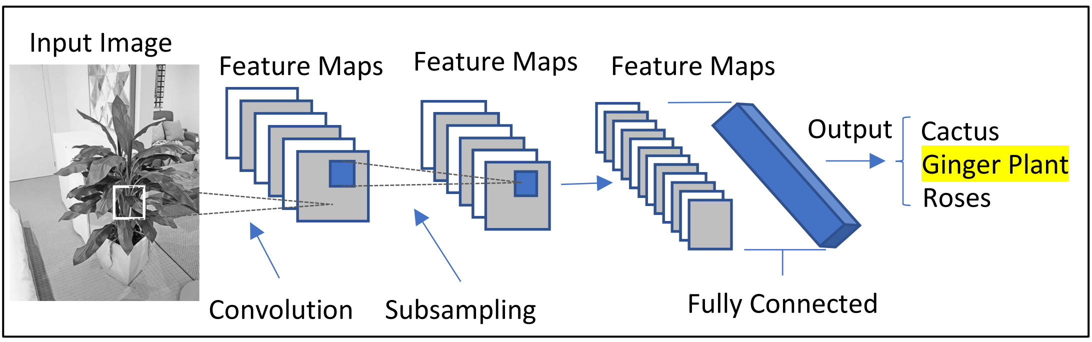



Figure 2.1: Use of CNN in Object Detection

## 2.3 YOLO (You Only Look Once)

### 2.3.1 YOLO's Introduction

YOLO is an object detection algorithm introduced by Redmon *et al.* (2016). YOLO is currently the most popular real-time object detector due to its lightweight network architecture, effective feature fusion methods and accurate detection results, The most widely accepted algorithms are YOLOv5 and YOLOv7 in terms of current usage (Lou, et al., 2023). The YOLOv5 uses deep learning technology for real-time and effective object detection, with improvements in model structure, training strategy, and overall performance. Unlike region proposal networks (R-CNN) or sliding windows to identify potential objects in an image, YOLO changed the approach to a single regression task, making predictions for bounding boxes and class probabilities directly from full images in one evaluation. However, it still has some limitations in detecting small object and dense object detection, along with complex situations such as occlusion and pose change.

YOLO uses Convolutional Neural Networks (CNNs) as the core of its architecture. YOLO is built on a CNN that processes the input image in a single pass to detect objects and their bounding boxes. The CNN extracts feature from the image, which are then used to predict the presence of objects, their locations, and class probabilities. The architecture of YOLO typically includes multiple convolutional layers followed by fully connected layers. The convolutional layers are responsible for feature extraction, where filters learn to detect patterns such as edges, textures, and shapes that are indicative of objects in the image. The fully connected layers then interpret these features to output the final predictions for object detection.

### 2.3.2 Evolution of YOLO Versions

Since it was first launched, YOLO has gone through multiple versions, with each one enhancing its predecessor's accuracy, speed, and ease of use. YOLOv2 proposed method by implementing methods such as batch normalization, anchor boxes,

and enhancing the feature extraction network. This help improved performance on different benchmarks compare to its predecessor. The architecture of YOLOv3 and YOLOv4 was improved by adding deeper networks, residual connections, and enhanced loss functions.

YOLOv5 and YOLOv8 which is not developed by the original creators, is developed to enhance the algorithm for improved detection speeds and accuracy. Additionally, these versions have increased YOLO's accessibility by providing pre-trained models and user-friendly frameworks to increasing its usability in different fields (Hussain, 2023).

YOLOv7 proposed a novel training strategy, Trainable Bag of Freebies (TBoF), which significantly improves the accuracy and generalization ability of the object detector. However, it requires more computational resources and training time to achieve the best performance, and its performance can degrade in some cases due to the training data, model structure, and hyperparameters (Lou, et al., 2023).

YOLOv8 uses Anchor-Free instead of Anchor-Base for improved performance which allows for dynamic "TaskAlignedAssigner" for matching strategy. It calculates the alignment degree of Anchor-level for each instance using Equation (). The algorithm selects (m) anchors with the maximum value (t) in each instance as positive samples and selects the other anchors as negative samples, then trains through the loss function. After these improvements, YOLOv8 is 1% more accurate than YOLOv5, making it the most accurate detector so far (Lou, et al., 2023).

$$t = s^{\alpha} \times u^{\beta} \quad (2.1)$$

where
$s =$ classification score
$u =$ IOU value
$\alpha$ and $\beta =$ weight hyperparameters

### 2.3.3 YOLOv8 Object Detection Mechanism

YOLOv8, published in 2023, combines the best of many real-time object detectors, adopting the idea of CSP from YOLOv5 (Wang, et al., 2020), feature fusion method (PANFPN) (Lin, et al., 2017), and SPPF module. Its main improvements include a brand new SOTA model, a detection head part that uses the current popular method of separating the classification and detection heads, and the use of BCE loss for classification and CIOU loss + DFL for regression (Lou, et al., 2023). The network quickly focused on the location distribution close to the object location, with probability density as close as possible to that the location, as shown in Equation (). YOLOv8 is also extensible and can support previous versions of YOLO, making it easy to compare the performance of different versions.

$$DFL_{(s_i,s_i+1)} = -\big((y_{i+1} - y)\log(s_i) + (y - y_i)\log(s_{i+1})\big) \qquad (2.2)$$

where

$s_i$ = output of sigmod for the network

$y_i$and $y_{i+1}$ = interval orders

$y$ = label

### 2.3.4 YOLOv8 Architecture and Segmentation Mechanism

While YOLO is primarily an object detection algorithm, its architecture can be adapted for segmentation tasks with YOLOv8. The segmentation classifies each pixel in the image, which is more complex than simply detecting objects and drawing bounding boxes around them. The architecture of YOLOv8 combine the convolutional neural networks (CNNs) and feature pyramid networks to capture both global context and fine-grained details. This multi-scale feature extraction process allows YOLOv8 identifying accurately object boundaries and generating high-quality segmentation mask (Terven, et al., 2024).

YOLOv8 implements segmentation by integrating a dedicated segmentation header into its architecture. This head is responsible for predicting masks for each detected object and refining the bounding box predictions to include detailed shape and area information. This simultaneous prediction of masks and bounding boxes allows YOLOv8 callable in real-time processing while providing a more comprehensive analysis of the scene (Wu, et al., 2024).

### 2.3.5 Advantages of YOLOv8 Segmentation

One of the standout features of YOLOv8 Segmentation is its ability to process images and videos in real-time. This capability is suitable for applications that require immediate feedback, such as autonomous vehicles, surveillance systems, and robotics. The model's architecture is optimized to ensure rapid detection and segmentation without sacrificing accuracy. As it excels in accurately detecting and identifying objects make it suitable use in complex scenarios involving small objects or occlusions,

The YOLOv8 segmentation model has been successfully adapted for applications ranging from agricultural monitoring to medical diagnostics and industrial inspections. In the field of agriculture, the YOLOv8-seg model, enhanced with Ghost and BiFPN modules, achieved an 86.4% Dice score in segmenting plant leaves, which outperforming existing methods (Wang, et al., 2024). Additionally, A modified YOLOv8-segANDcal model improved detection and segmentation of soybean radicles by 2% and 1% in mAP, facilitating rapid crop variety selection (Wu, et al., 2024).

In the field of medical Imaging, YOLOv8 demonstrated high efficacy in segmenting polyps in colonoscopy images, achieving a Dice score of 0.919, which aids in colorectal cancer diagnosis (Sandro Luis de, et al., 2024). While the YOLOv8 used in corrosion detection, YOLOv8's single-pass detection method allows for efficient corrosion segmentation in industrial imagery, enhancing maintenance strategies (R S, et al., 2024).

## 2.4 Faster R-CNN

Faster R-CNN is an object detection model introduced by Ren *et al.* (2017). Faster R-CNN an extension of the R-CNN (Region-based Convolutional Neural Network) framework and it is aiming to improve the detection speed without compromising accuracy. Faster R-CNN has been used in different sectors, such as agriculture for detecting and harvesting fruits with deep learning (C, et al., 2022), as well as in Internet of Vehicles (loV) for instant object detection in intelligent transportation systems (Zineb, et al., 2023). It has also been evaluated in systematic literature reviews while comparing it with other object detection algorithms like YOLO (Rashid & Fadzil, 2023).

Since Faster R-CNN has achieved near real-time processing speed with the used of very deep networks. However, the computational bottleneck problem come out from the time spent on generating region proposals which is an important in state-of-the-art detection systems. In order to solve this, there are difference method explored to leverage deep networks for the localization of class-specific of class-agnostic bounding boxes. One of the methods involves using the Multi-Box methods where a regions proposals are generated directly from the network's last fully connected (fc) layer (Erhan, et al., 2013). This help predict multiple bounding boxes simultaneously and is used for object detection within the Faster R-CNN framework. Additionally, another method that achieves high accuracy in real-time object detection with high processing speed is the YOLO (You Only Look Once) algorithm. As one of the YOLO algorithm, YOLOv5, stands out for its high speed and accuracy, hitting 69 frames per second on the COCO dataset and maintaining a mean Average Precision (67%) equivalent to SE-YOLOv3 (Reswara , et al., 2023).

## 2.5 Image enhancement

Due to variation to the appearance of the plants due to environmental factors such as soil type and condition, exposure to sunlight, and water availability. This causes the images having shadows or illumination effect and affect the performance of leaf region

identification. Therefore, improving images is an important initial stage in many tasks, such as extracting plant leaves. The methodology of enchanting an image is shown in Figure 2.2.

Figure 2.2: Process of Image Enhancement

In object extraction, image enhancement is required to minimize the impact of shadows and illumination on the identification of target regions. The V (Value) plane in the corresponding HSV (Hue, Saturation, Value) image represents the brightness of an image (Ganesan, et al., 2014). Pre-processing the V plane in the HSV colour space can help decrease the illumination effect, ultimately enhancing the segmentation accuracy.

The suggested approach starts by transforming the plant's RGB image into an HSV image. Next, the V plane is analysed to determine the skewness ($S_k$) using the Equation  and the probability distribution of the V plane. If the skewness is positive, meaning there is shadow, the V plane needs to be enhanced to eliminate the illumination effect (Praveen & Domnic, 2019). On the flip side, an image with excessive brightness will result in a negative skewness value, showing that the distribution is leaning towards the right.

$$S_k = \frac{E\left(x_{ij} - \mu\right)^3}{\sigma^3} \tag{2.3}$$

where

$x_{ij}$ = value of $(i, j)^{th}$ pixel in V plan of HSV image before image enchancement

$E$ = expected value

$\mu$ = mean value

$\sigma = \text{standard deviation}$

In statistical modelling of contrast enhancement, the statistical properties of an image can be analytically calculated using the probability distribution function of an image (Sahar Jorjandi *et al.*, 2021). The probability distribution is a function that gives all the possible values and likelihoods of a random variable within a range. In the proposed work, the distribution of brightness (V plane) is assumed to be in a Weibull distribution. To remove the over brightness or shadow effect in the image, the skewness of the brightness distribution should be made symmetric to allows for the removal of over brightness or shadow effect in the image, further enhancing the image and improving the segmentation accuracy.

## 2.6 Leaf Segmenting

Segmenting plants is an important part of examining plant characteristics like height, leaf area, colour, texture, and shape. It required isolating the plant area from the surrounding background within an image. There are different image segmentation techniques such as thresholding, edge detection, and segmentation methods based on machine learning can accomplish this (Manjula, 2017).

Thresholding is a straightforward technique where a threshold value is used to differentiate the plant area from the background by identify the intensity values of the pixels (Al-amri, et al., 2010). This technique works best for photos with distinct contrast between the plant and the background. Plant segmentation can also use edge detection as a technique. This process includes identifying the plant region's edges in the image and distinguishing it from the background. This technique is beneficial for images with clearly defined boundaries in the plant area (Salman, 2006).

Plant segmentation can also be accomplished using segmentation methods based on machine learning. These techniques require the machine learning model to be trained on a dataset of images where plant regions are labelled. The model can be

utilized to separate new images by forecasting the area of the plant using the image's characteristics (Lee, et al., 2018).

After segmenting the plant region, different characteristics can be extracted in order to analyse the plant. Plant height, leaf area, colour, texture, and shape are some of the characteristics mentioned. Height of plants can be assessed through image processing methods like distance transformation or edge detection. Contour analysis and area calculation methods can be used to determine leaf area. Characterizing the colour distribution of the plant can involve extracting colour features like mean colour, colour histograms, or colour moments. Texture features such as Haralick texture features, Gabor features, and Local Binary Patterns (LBP) can be calculated to represent the texture patterns found in the plant areas (Porebski, et al., 2008)

## 2.7 Leaf extraction and classification

Leaf extraction and classification are fundamental tasks in plant species identification such as agriculture, botany, and environmental science. Leaf extraction and classification involve the use of different feature extraction methods to classify the species based on different leaf features such as including shape, texture and venation. This literature review explores methods and techniques employed in leaf extraction, particularly focusing on shape features and graph-based algorithms for segmentation.

Shape features are commonly used in plant leaf classification. a study show that leaf shape features have been chosen and tested in almost 62.5% of plant identification studies, much exceeding the use of other features. This is because they are the easiest and most obvious features for distinguishing species, particularly for non-botanists who have limited knowledge of plant characters (Lee *et al.*, 2017). However, the performance of these approaches is highly dependent on a chosen set of hand-engineered features, which are liable to change with different leaf data and feature extraction techniques, confounding the search for an effective subset of features to represent leaf samples in species recognition studies.

One of the methods involves using the enchanted HSV (Hue, Saturation, Value) images for leaf segmentation (Praveen & Domnic, 2019). The process involves segmenting the leaf region based on the V plane while maintaining robustness to reflections and shadows. A graph-based algorithm is employed for segmentation, enhancing accuracy while minimizing computational complexity. The algorithm constructs a graph representing the enhanced HSV image, with nodes representing pixels and edges defining relationships between neighbouring pixels or between pixels and source/terminal nodes.

Edge costs are determined based on prediction parameters derived from pixel values in the HSV image (Lee, et al., 2017). These costs guide the segmentation process, distinguishing leaf pixels from background or non-leaf pixels. The segmentation algorithm iteratively explores the graph to identify leaf regions, facilitated by search trees originating from source and terminal nodes

Furthermore, post-segmentation refinement is conducted to eliminate non-leaf regions such as light reflections, yellow soil, and mosses. Conversion to RGB, CMYK, and Lab colour spaces enables effective discrimination between leaf and non-leaf elements (Lee, et al., 2017). Threshold values are empirically determined for each dataset, ensuring accurate removal of undesirable elements from the identified leaf regions.

CHAPTER 3

# METHODOLOGY

## 3.1 Introduction

The methodology in integrating deep learning algorithm to monitor the ginger plants involves multiple steps which is necessary for obtaining the precise and effective detection outcomes. The process is shown as below:

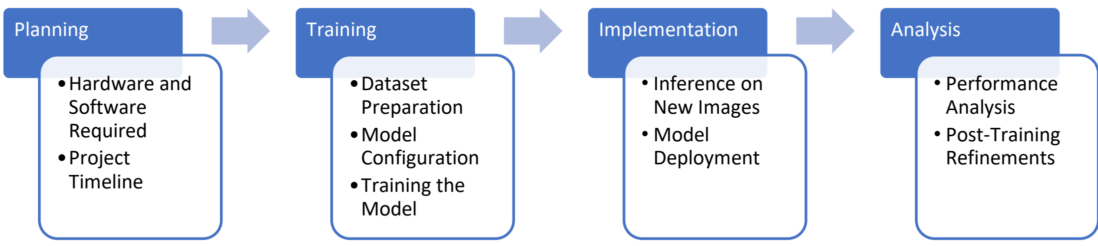



Figure 3.1: Process Flow of Detecting the Ginger Plant

## 3.2 Planning

### 3.2.1 Hardware Required

#### 3.2.1.1 Image Capture

Lenovo 300 FHD webcam will be used to capture images for the dataset. This webcam provides full HD resolution to ensure high-quality input data for the training process.

#### 3.2.1.2 Inference Hardware

The local machine should have a GPU compatible with CUDA 11.7 to enable accelerated inference for running models like YOLOv8 and the depth estimation model efficiently.

### 3.2.2 Software Required

#### 3.2.2.1 Training Environment

Google Colab will be used for training model. Google Colab offers the advantage of using high-performance GPUs in the cloud, which is ideal for deep learning tasks.

#### 3.2.2.2 Inference Environment

The local environment will use Python 3.11.7, which is compatible with the latest deep learning libraries. The CUDA 11.7 should also be installed to ensures GPU acceleration for running deep learning models, which is crucial for real-time inference.

#### 3.2.2.3 Model Required in The System

- **YOLOv8 Segmentation for Ginger Plant Classification**: The YOLOv8 model will be trained and utilized for segmentation tasks to classify whether a plant is a ginger plant.
- **YOLOv8 Segmentation for Healthy and Unhealthy Ginger Plant Leaves Classification**: Another YOLOv8 model will be trained deployed for segmenting and classifying ginger plant leaves as either healthy or unhealthy.
- **Depth Estimation Model**: The Intel DPT-Large model will be used for estimating the depth of the plants.

### 3.2.3 Project Timeline & Resource allocation

The project will begin with data collection, where images will be captured using the webcam to create datasets for training the YOLOv8 model. Once the dataset is ready, the model training phase will start using Google Colab for training because Google Colab provides GPU resources. The training will involve running multiple epochs with evaluations to monitor and optimize the model's performance. Following the completion of training, the project will transition into setting up the local inference environment. This setup will involve installing and configuring all necessary software dependencies to ensure success of inference process.

In terms of resource allocation, significant time will be dedicated to capturing and annotating the dataset, as this step is critical to the overall success of the project. The training phase will rely on Google Colab's GPU resources to accelerate the process and achieve faster results. For the inference and testing phase, the local machine will be optimized with the CUDA configuration to enable faster processing and testing of the model.

Table 3.1: Gantt Chart for FYP1

| **Task** | **Week** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
| Project Title Decision | ■ | ■ | | | | | | | | | | | | |
| Introduction | | | ■ | ■ | ■ | | | | | | | | | |
| Review on Literature | | | | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | | | |
| Methodology | | | | | | | | ■ | ■ | ■ | ■ | ■ | | |
| Planning | | | | | | | | | | ■ | ■ | ■ | ■ | ■ |

Table 3.2: Gantt Chart for FYP2

| **Task** | **Week** | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
| Develop Ginger Plant Detection | ■ | ■ | ■ | ■ | ■ | ■ | | | | | | | | |
| Develop Plant Leaf Detection | | | ■ | ■ | ■ | ■ | ■ | ■ | | | | | | |
| Develop Depth Estimation Model | | | | | | ■ | ■ | ■ | ■ | | | | | |
| Evaluate Model | | | ■ | ■ | ■ | ■ | ■ | ■ | ■ | | | | | |
| Organize Results | | | | | | | | ■ | ■ | ■ | ■ | | | |
| Discussion on Results | | | | | | | | | | | ■ | ■ | ■ | |
| Conclusion and Recommendation | | | | | | | | | | | | | | ■ |

## 3.3 Training

### 3.3.1 Dataset Preparation for YOLOv8 model training

The training process began with the preparation of the dataset. Images were captured using the webcam to create datasets. These images were then annotated using tools "Roboflow", marking each image for segmentation and object detection to identify the ginger plants and their leaves accurately. To enhance the model's ability to increase the diversity of the dataset and help the model learn more robust features, the data augmentation techniques were applied such as transformations, rotation, scaling, flipping, and colour adjustments.

Two YOLOv8 models were trained for different purposes. The first model was trained to detect and classify whether a plant is a ginger plant or not, while the second model was trained to detect the leaves of the ginger plant and classify them as either healthy or unhealthy. During the training phase of these two YOLO models, key hyperparameters such as learning rate, batch size, and the number of epochs were tuned to optimize the model's performance. An early stopping mechanism was implemented to prevent overfitting, ensuring the model did not learn to perform well only on the training data. The table below show the hyperparameters use in training the plant and leaf segmentation.

Table 3.3: Hyperparameters use in Training Phase

| **Task** | Segment |
|---|---|
| **Mode** | Train |
| **Pretrained weight** | yolov8n-seg |
| **Device** | Google Colab's GPU |
| **Epochs** | 40 |
| **Learning Rate** | 0.001 |
| **Batch Size** | 16 |
| **Imgsz (Image Size)** | 640 |

Training was conducted using Google Colab's GPU resources, which provided the necessary computational power. The YOLOv8 model underwent multiple epochs of training, with its performance evaluated at each stage. Throughout this process, model checkpoints were saved to allow the best-performing model can be retrieved if needed.

A portion of the dataset was reserved as a validation set and test set, which was used to evaluate the model's performance throughout the training process. After training, the model was tested on a separate test set to assess its effectiveness, with metrics such as Intersection over Union (IoU) and accuracy calculated to evaluate segmentation and detection performance.

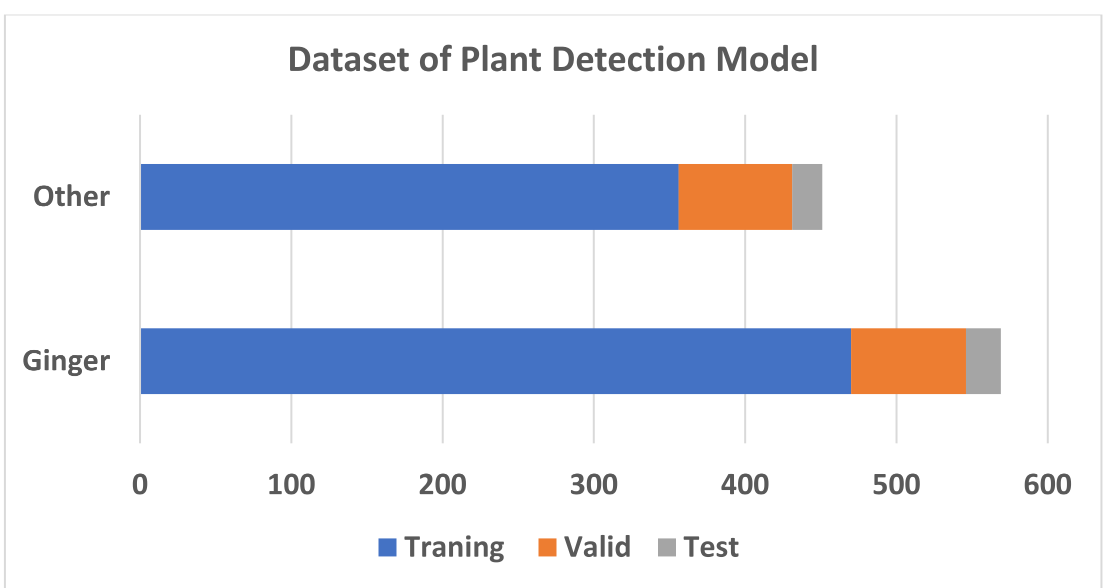



Figure 3.2: Dataset of Plant Detection Model

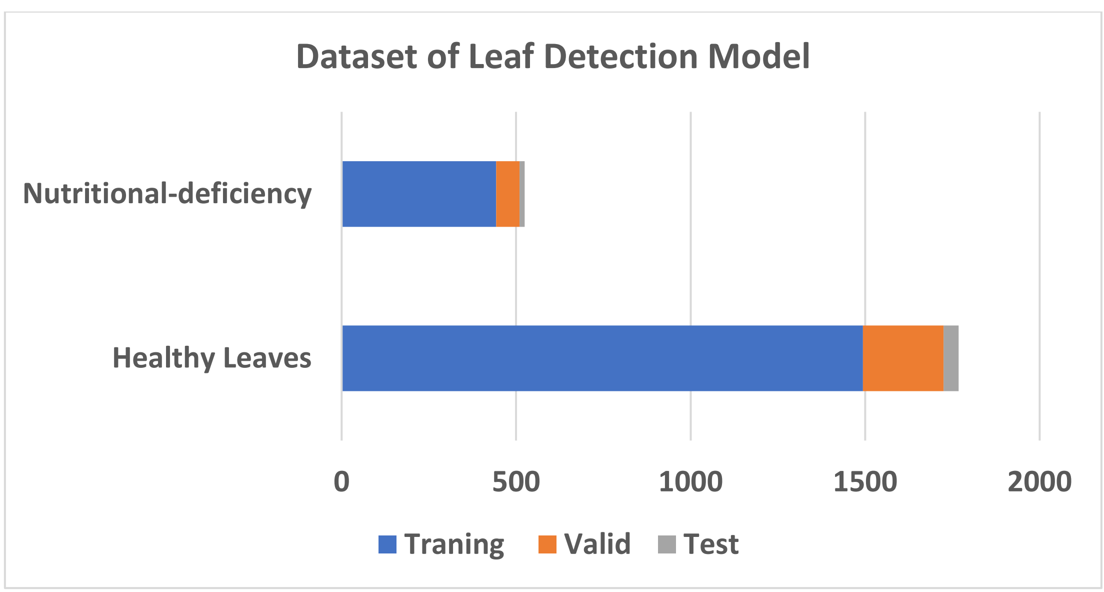



Figure 3.3: Dataset of Leaf Detection Model

### 3.3.2 Dataset Preparation for Depth Estimation Model Training

To estimate the depth of the ginger plants in order to calculating their height, the Intel DPT-Large model was integrated into the workflow. The dataset for training the depth estimation model was carefully curated to ensure accurate and reliable depth predictions. The images were captured using the Lenovo 300 FHD webcam, focusing on different distances to capture the full range of depth variations in the ginger plants.

To create the ground truth for depth estimation, the dataset was annotated with depth information corresponding to each image. This annotation process involved using a combination of sensor data and manual labelling to accurately represent the distance of the plants from the camera. The data was then pre-processed to match the input requirements of the Intel DPT-Large model, including resizing, normalization, and augmentation to improve the model's ability to generalize.

The Intel DPT-Large model was chosen for its ability to deliver high-quality depth predictions, particularly in complex environments. During training, the model was optimized using a custom loss function that minimized the difference between the predicted and actual depth values. Hyperparameters such as learning rate, batch size,

and the number of epochs were carefully tuned to achieve the best possible performance.

The training process was conducted using powerful computational resources to handle the large and complex dataset. As with the YOLOv8 model training, 20% of the dataset was reserved as a validation set to monitor the model's performance throughout the training process. This validation ensured that the model was learning effectively and that any issues such as overfitting were addressed promptly.

After the initial training phase, the model's performance was evaluated using key metrics such as mean absolute error (MAE) and root mean square error (RMSE), which provided insights into the accuracy of the depth predictions. Based on these results, the model was fine-tuned by adjusting hyperparameters and retraining with an augmented dataset. This fine-tuning aimed to enhance the model's ability to accurately estimate the depth of ginger plants under various conditions, ensuring its effectiveness in real-world applications.

## 3.4 Implementation

### 3.4.1 Inference on New Images

Following the evaluation, the models were deployed for inference on new images. The inference process involved applying the trained YOLOv8 model to detect and segment ginger plants and their leaves in unseen data. The depth estimation model was also used to measure the distance of the detected plants from the camera, specifically focusing on the height and area calculation of the ginger plant's leaves.

The implementation was tested across various scenarios to ensure the models performed well under different conditions. The results from these tests were recorded and analysed to determine the consistency and reliability of the models in practical applications.

### 3.4.2 Model Implementation

This section describes the real-time monitoring system and detection system of the deployed system. The Figure 3.4 shows the overview of implementation phase of the system.

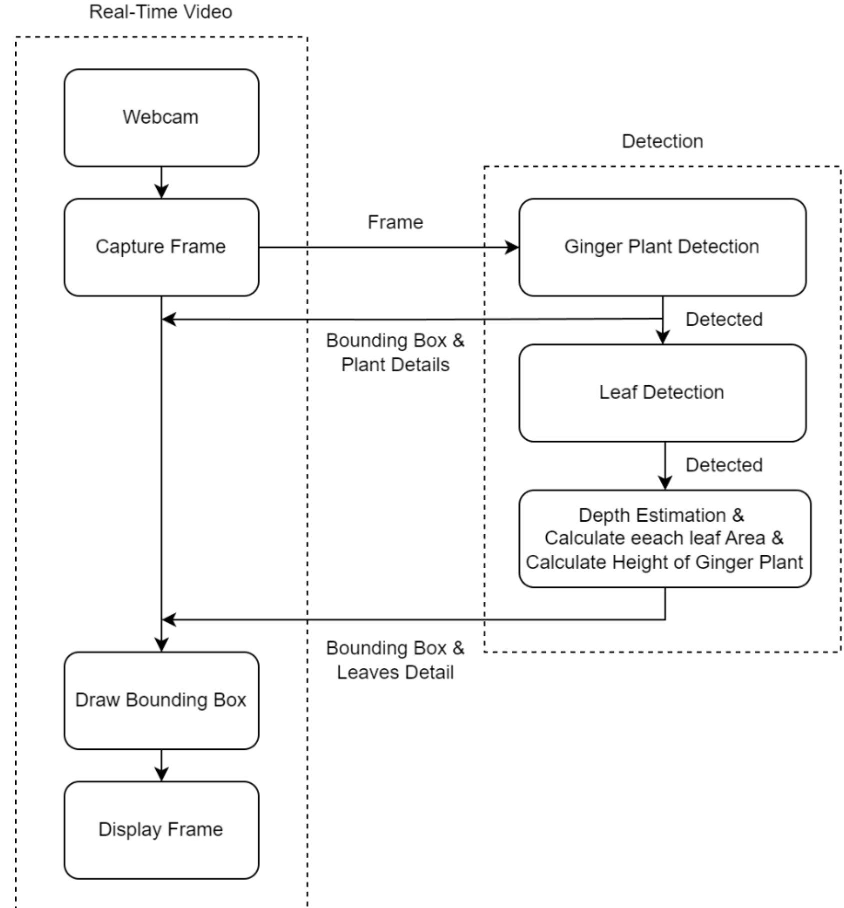



Figure 3.4: Overview of Model Implementation Phase

The trained model was deployed locally using Python 3.11.7 and CUDA 11.7 for real-time inference. The inference process is optimized by adjusting the batch sizes, image resolution, and any post-processing that the system can handle real-time input from the webcam.

Two trained YOLOv8 models and depth estimation were deployed, one for detecting the ginger plant and the other for segmenting its leaves. Additionally, the depth estimation model was deployed to calculate the height of detected ginger plant

and its leaves. To enhance performance, the inference process was optimized by adjusting batch sizes, image resolution, and post-processing steps.

## 3.5 Analysis

### 3.5.1 Performance Analysis

In the Analysis phase, the trained YOLOv8 model's performance will be examined by comparing the evaluation metrics such as precision, recall and mAP against established benchmarks. This comparison highlighted the strengths and weaknesses of the YOLOv8 model in detecting and segmenting ginger plants. The mathematical formula for these metrics is provided.

$$Acccuracy = \frac{TP}{TP + FN + FP + TN} \tag{3.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

$$Recall = \frac{TP}{TP + EN} \tag{3.3}$$

$$F1\ score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3.4}$$

where

$TP$ = True positive

$TN$ = True negative

$FP$ = False positive

$FN$ = False negative

$$mAP = \frac{1}{N}\sum_{i=1}^{N}\overline{P_i} \quad (3.5)$$

where

$\overline{P_i}$ = Average precision for a given sample $N$

$N$ = Sample

Furthermore, multiple curves will be plotted to determine the selection on best threshold confidence. The F1-confidence curve plots the F1 score, the harmonic mean of precision and recall, against different confidence thresholds. The F1 score is a single metric that balances precision and recall to measure a model's performance. The precision-confidence curve shows how the precision changes with confidence. Precision measures the proportion of true positive detections among all detections made by the model. This helps obtain the threshold between making correct positive predictions and avoiding false positives.

The precision-recall curve is used to evaluate the model's effectiveness, especially in cases of imbalanced datasets. The precision-recall curve plots precision against recall for different confidence thresholds. Finally, the recall-confidence curve shows how recall, or the model's ability to detect true positives, varies with confidence, indicating how sensitive the model is to detecting objects as the threshold changes. The recall-confidence curve shows how recall varies with the confidence threshold. Recall measures the proportion of true positive detections among all actual positive instances.

The accuracy of depth estimation was also compared with ground truth measurements to assess the precision of the height and area calculations of the leaves.

Mean Absolute Error (MAE) is calculated as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_i}| \quad (3.6)$$

The Root Mean Squared Error (RMSE) is given by

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \quad (3.7)$$

where

$y_i$ = actual depth

$\hat{y}_i$ = predicted depth

$n$ = number of samples

## 3.6 Cost Estimation

This section gives the project's cost estimation to demonstrate the project's budget and ensure that there is enough funding for the system's development. The table provided indicates the total cost estimate.

Table 3.4: Cost Estimation of Project Materials

| Item | Cost (RM) |
|---|---|
| **Hardware** | |
| Lenovo 300 FHD webcam | 170.00 |
| **Material** | |
| Ginger Plants | 25.00 |
| Gingers | 10.00 |
| Soil | 10.00 |
| Pot | 20.00 |
| Total Estimated Cost | 235.00 |

# CHAPTER 4

# RESULTS AND DISCUSSIONS

## 4.1 System Interface Results

In this section, the result of the created system was discussed. The evaluation of YOLOv8's results and performance were conducted. During this phase, the system that was developed was evaluated with various ginger plants. The figure below shows the real-time plant monitoring system interface.

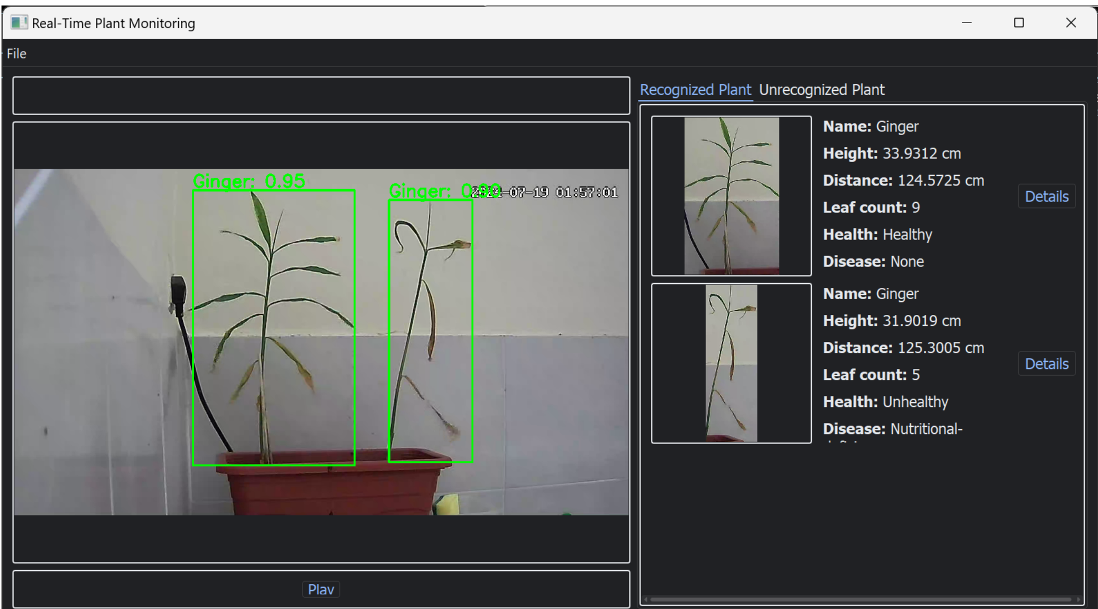



Figure 4.1: Real-Time Plant Monitoring System Interface

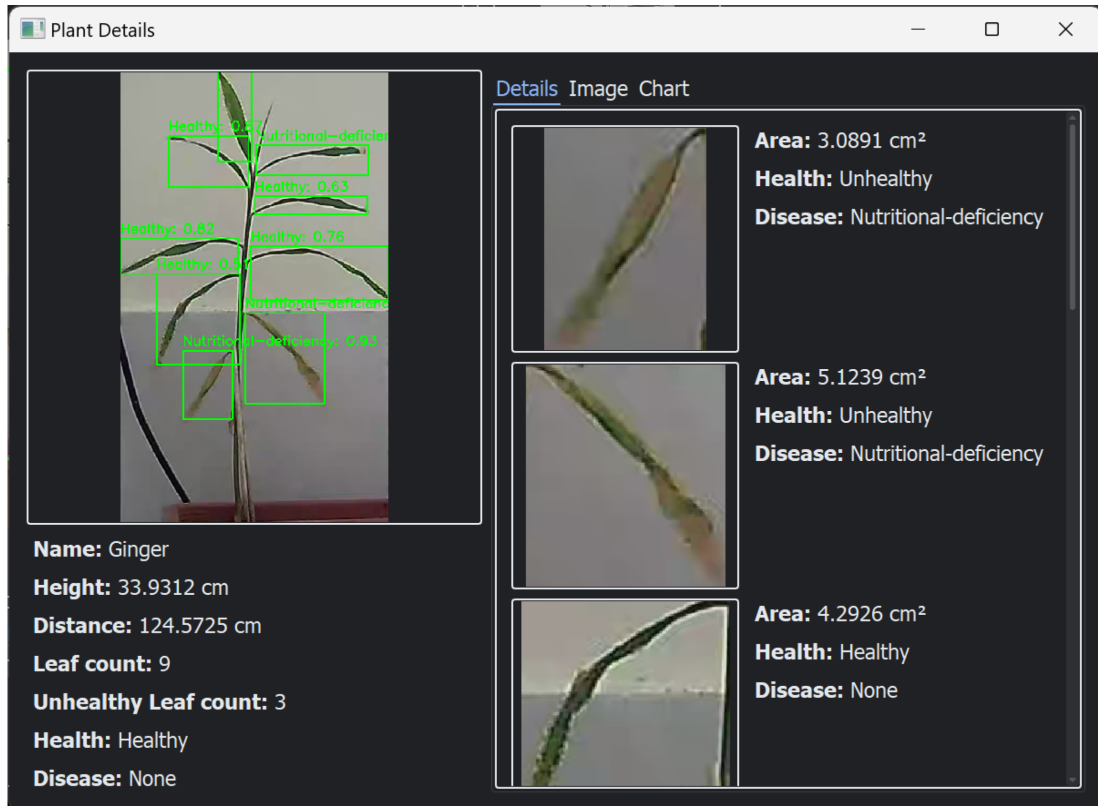



Figure 4.2: Real-Time Plant Monitoring System Interface

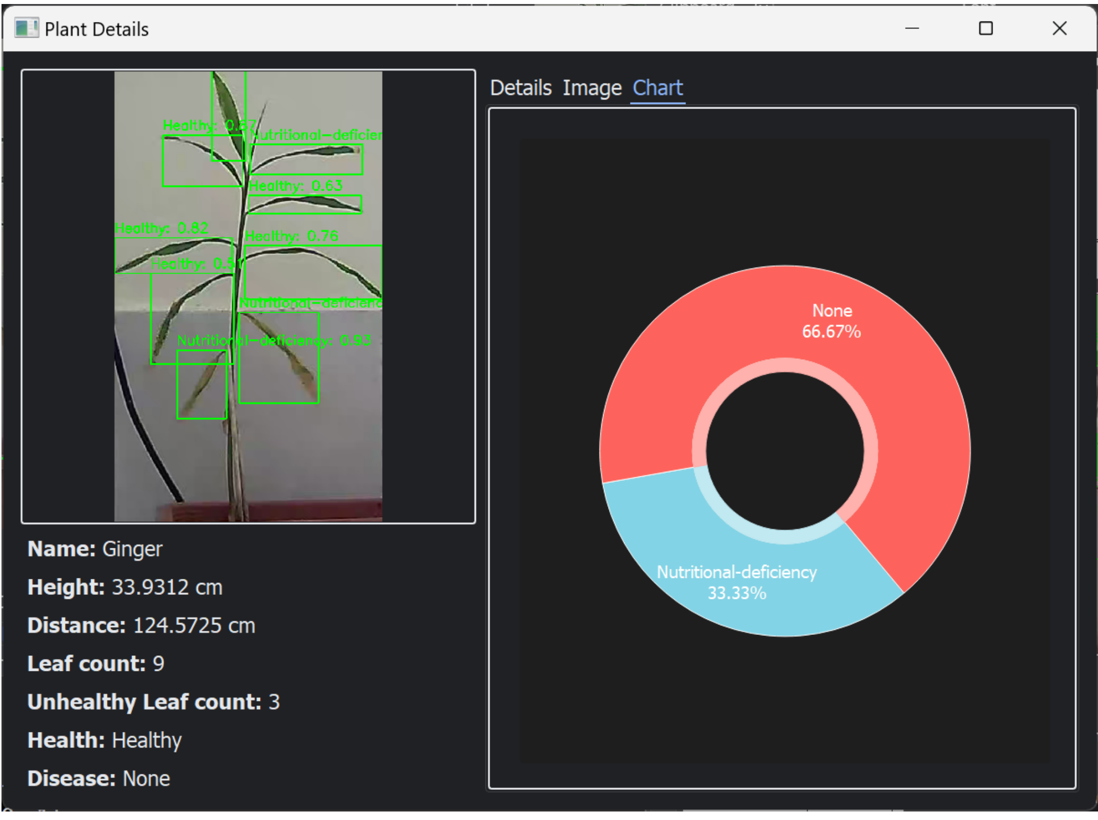



Figure 4.3: Real-Time Plant Monitoring System Interface

## 4.2 Ginger Plant Detection Using YOLOv8 Model

The first model was trained using YOLOv8 for ginger plant detection. This model segment ginger plants from the background and other types of plants, labelling the detect ginger plant as "ginger", while labelling other plant as "other". The performance of the model was evaluated using metrics such as Precision, Recall, and F1 score.

### 4.2.1 Example of Ginger Plant Detection Results

The result of training the plant detection model was shown in Figure 4.4. In the figure, blue bounding boxes indicate detected ginger plants, while cyan bounding boxes correspond to other plants.

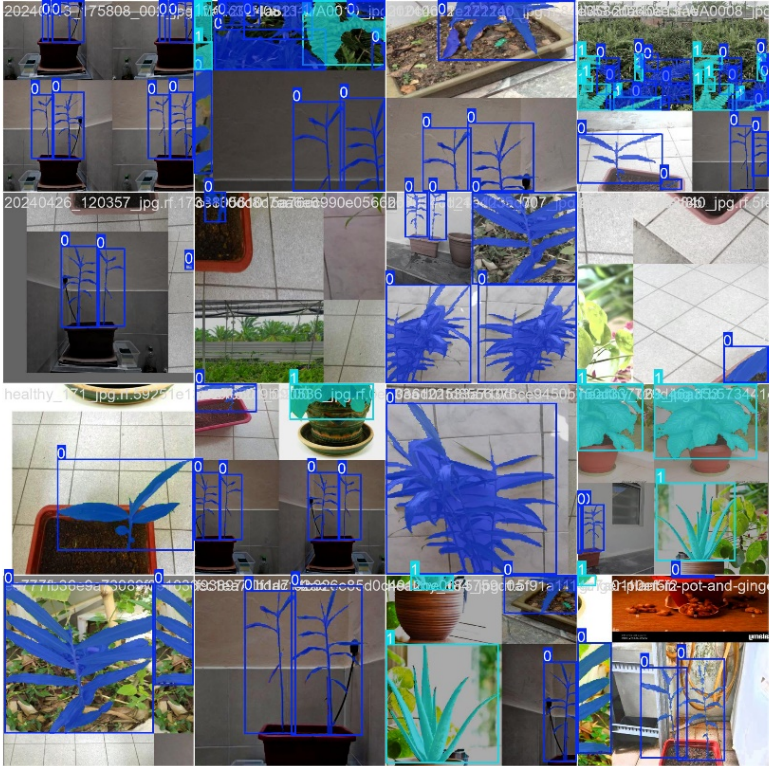


Figure 4.4: Labels in Training Process of Plants Detection

Table 4.1: Example Testing Results of YOLOv8 Segmentation Ginger Plant Detection Model

| No | Original Image | Plot | Extracted Region |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |

### 4.2.2 Ginger Plant Detection Using YOLOv8 Model Performance

To evaluate the performance of the model, the precision, recall, and F1-score were calculated based on the detected and actual values and these metrics were plotted against each other to provide the accuracy of the model.

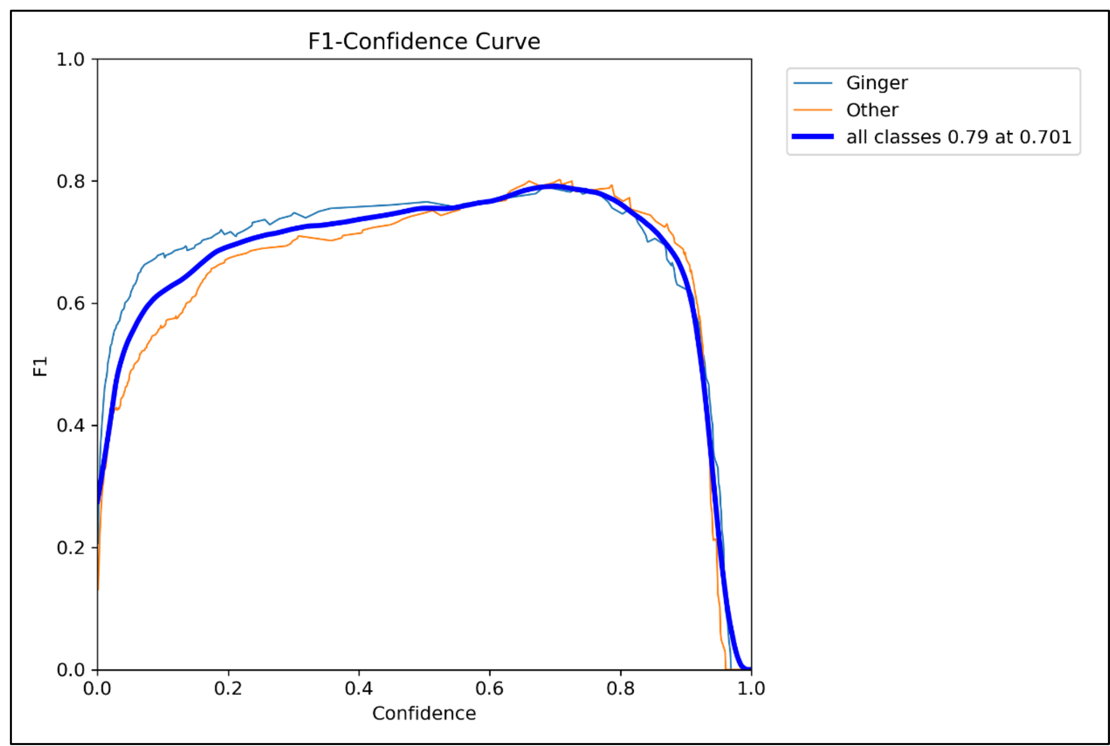



Figure 4.5: F1-Confidence Curve

Based on the curve in Figure 4.5, the model achieves an F1 score of 0.79 at a confidence threshold of 0.701 across all classes, indicating that the model maintains a strong balance between precision and recall when making predictions at this confidence level. Since the F1-confidence curve for ginger plants and other plants follows the overall curve, the model's average performance across all classes is consistent in detecting and classifying different plants. Therefore, the consistent performance across classes is a positive indicator that proves that it can correctly identify ginger plants. However, the curve shows that the trend declines sharply after the peak because beyond this threshold, the F1 score decreases as the model becomes more conservative, which may miss some true positives (ginger plants) in favor of increasing precision.

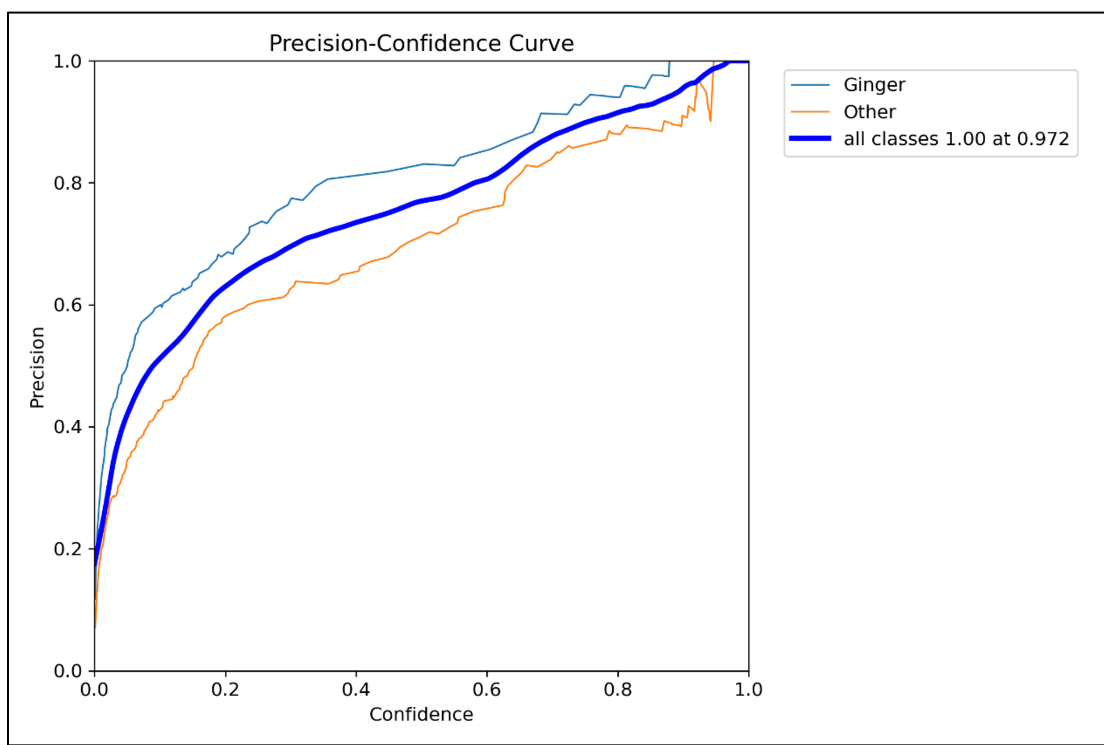



Figure 4.6: Precision-Confidence Curve

Based the curve in Figure 4.6, the model achieves a perfect precision score of 1.00 at a confidence threshold of 0.972 across all classes, indicating the model is highly confident about its predictions in detecting and classify the plant. It also indicates that all the predicted instances of plants match to the actual instances of plant, resulting it in an almost 100% accuracy. However, this high precision may cause the model only makes predictions when it is very high confidence, because at elevated confidence levels does not necessarily translate to overall effectiveness. While high precision is desirable, but it’s also important to balance this with recall, especially in agricultural applications where missing ginger plants or misclassifying other plants will causing consequences such as reduction losses.

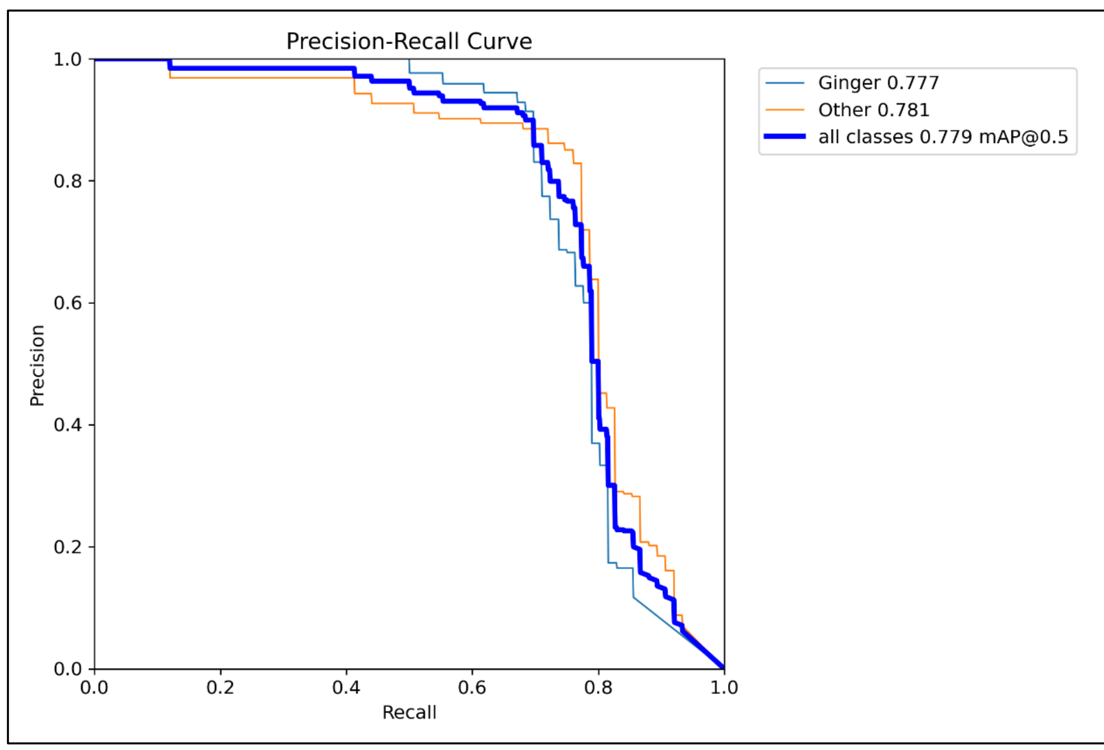



Figure 4.7: Precision-Recall Curve

Based on the curve in Figure 4.7, it provides how the model's performance compared to the ROC curve. A precision-recall score of 0.777 for detecting ginger plants (ginger) suggests that the model is effective in identifying ginger plants, with a good balance between precision and recall. The precision-recall score for detecting other plants (other) is slightly higher at 0.781, indicating that the model performs comparably well in detecting other plant types. The mean Average Precision (mAP) at 0.5 for all classes is at 0.79, indicating that the model has strong performance in plant detection.

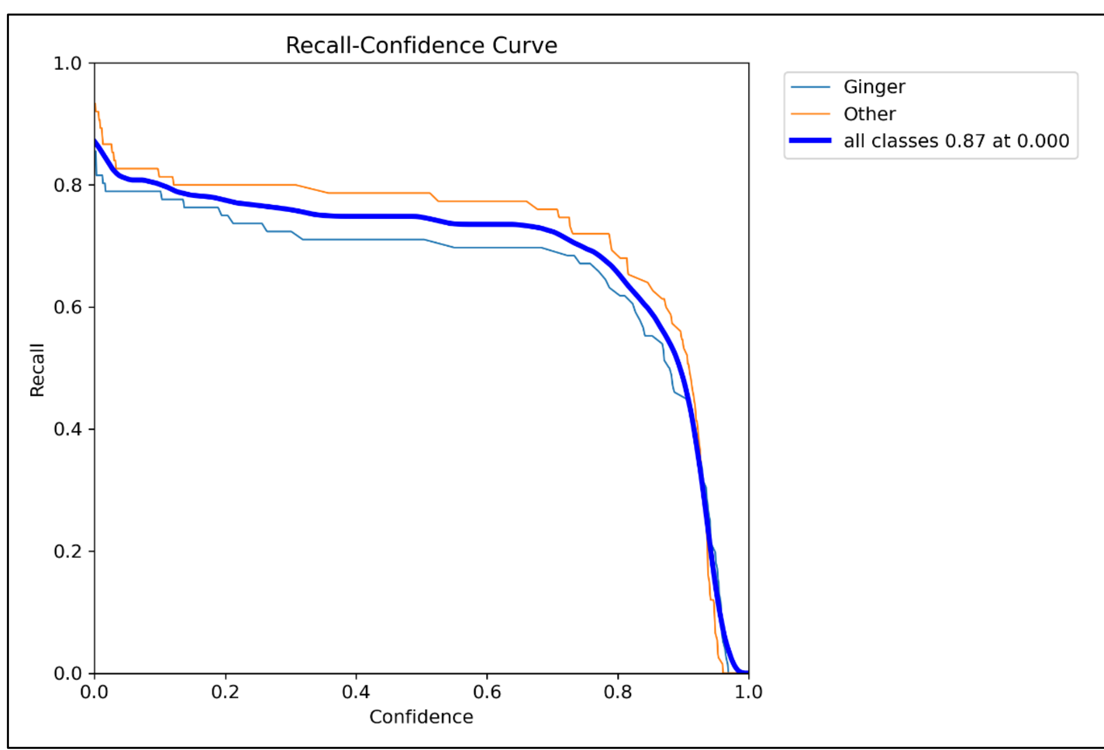



Figure 4.8 Recall-Confidence Curve

Based on the curve in the Figure 4.8, the model achieves a recall score of 0.89 across all classes at a confidence level of 0.000. This suggests that the model can identify a high proportion of actual ginger plants and other plants when it makes predictions. However, the low confidence threshold indicates that the model may lead to many false positives.

This curve shows how much recall is sacrificed when the model's confidence level increases. A steep decline in recall at a certain threshold indicates that beyond that threshold confidence level would reduce the model's ability to detect true positives. While a high recall score is beneficial, it is also required to analyse the trade-off between precision and recall. A model that predicts too many plants at low confidence may overwhelm users with false positives, leading to inefficiencies in agricultural monitoring. Given the observed decline in recall around the 0.8 confidence level, selecting the confidence at this level will be an effective balance that will predict and classify the plant accurately while minimizing false positives.

## 4.3 Leaf Detection and Health Classification Using YOLOv8

The second model, also based on YOLOv8, was trained to detect leaves on the ginger plant and classify them as healthy or unhealthy. The detect leaves while labelling healthy leaves as "Healthy" and classify unhealthy leaves to “Nutritional-deficiency”.

### 4.3.1 Example of Leaf Detection and Health Classification Results

The result of training the plant detection model was shown in Figure 4.9. In the figure, blue bounding boxes indicate detected healthy leaves, while cyan bounding boxes correspond to unhealthy leaves.

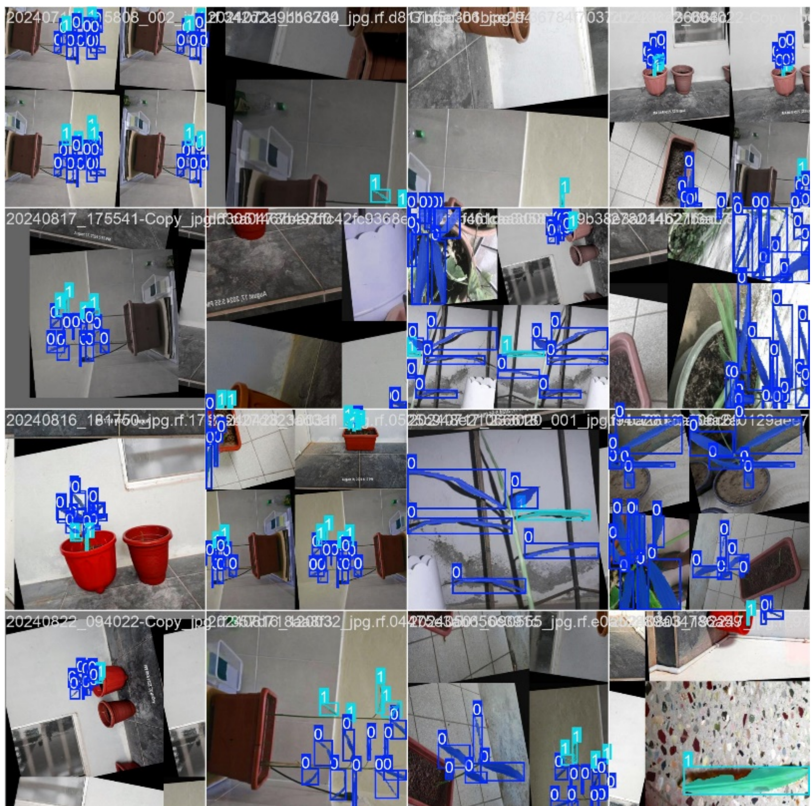


Figure 4.9: Labels in Training Process of Leaves Detection

Table 4.2: Example Testing Results of YOLOv8 Leaves Detection Model

| No | Original Image | Plot | Extracted Region |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |

### 4.3.2 Leaf Detection and Health Classification Using YOLOv8 Performance

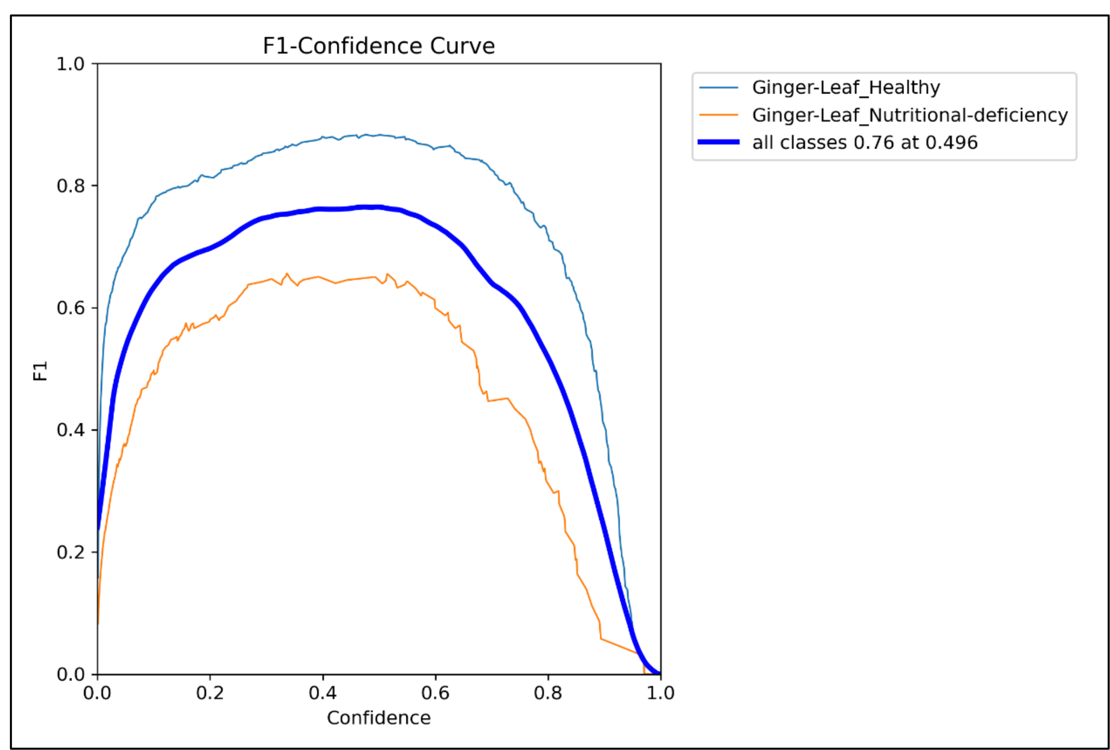



Figure 4.10: F1-Confidence Curve

Based on the curve in Figure 4.10, the curve shows that healthy leaves have a higher peak compared to unhealthy leaves. The model is more effective at detecting and classifying healthy leaves, as the curve shows a higher F1 score in detecting healthy leaves. However, the lower peak for unhealthy leaves indicates that the model is challenging in detecting and classifying the unhealthy leaves and leads to higher rates of false negatives or false positives. The peak F1 score for all classes is 0.76 at a confidence level of 0.496. Therefore, selecting this threshold will provide a good balance across all classes.

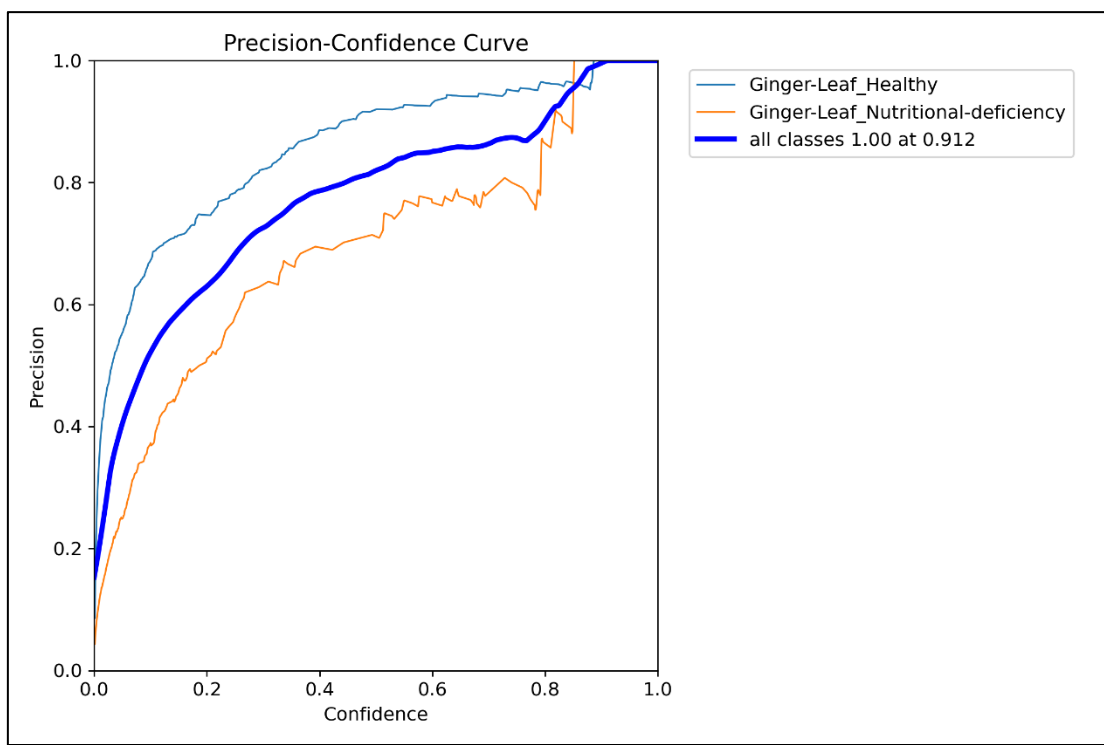



Figure 4.11: Precision-Confidence Curve

Based the curve in Figure 4.11, the model achieves a perfect precision score of 1.00 at a confidence threshold of 0.912 across all classes, indicating that when the model is confident in its predictions, it is highly accurate in classifying leaves correctly. At low confidence thresholds, the model will predict more objects, but some of these predictions may be incorrect and reduce precision. While high precision is desirable, it is essential to balance this with recall, particularly in agricultural applications where missing unhealthy leaves could have significant consequences. The model's performance at lower confidence levels should also be examined to ensure it can identify unhealthy leaves effectively.

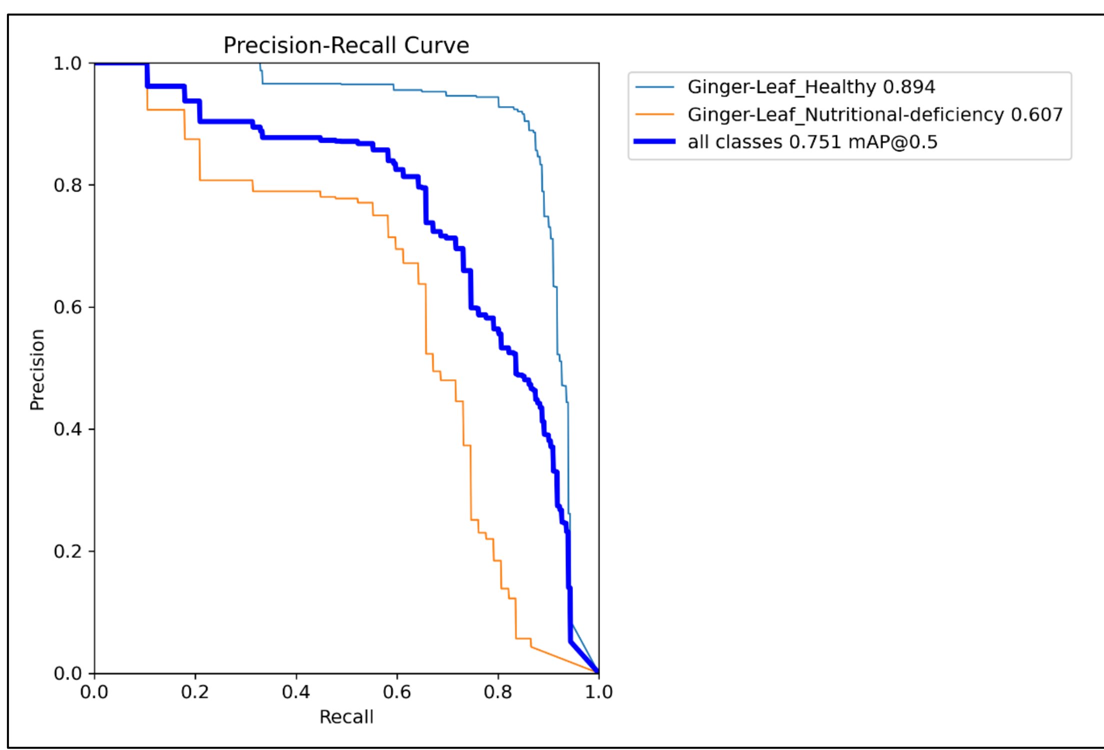



Figure 4.12: Precision-Recall Curve

Based on the curve in Figure 4.12, the higher precision and recall will have a curve that is close to the top-right corner of the graph. The shape of the curve is indicative of the model's performance, a steep drop-off indicates a point where increasing recall significantly reduces precision, which may suggest that the model is starting to predict more false positives. In detecting healthy leaves, the precision-recall score is 0.894 which means that the model is more accurate in identifying healthy leaves compared to detecting unhealthy leaves as the precision-recall score for unhealthy leaves is 0.607. Therefore, the model is less reliable in detecting unhealthy leaves, which could lead to missed opportunities for early intervention in crop management. The mean Average Precision (mAP) at 0.5 for all classes is reported at 0.751, indicating a good overall performance. However, the significant difference in scores between healthy and unhealthy leaves underscores the need for targeted improvements in the model's training and evaluation processes.

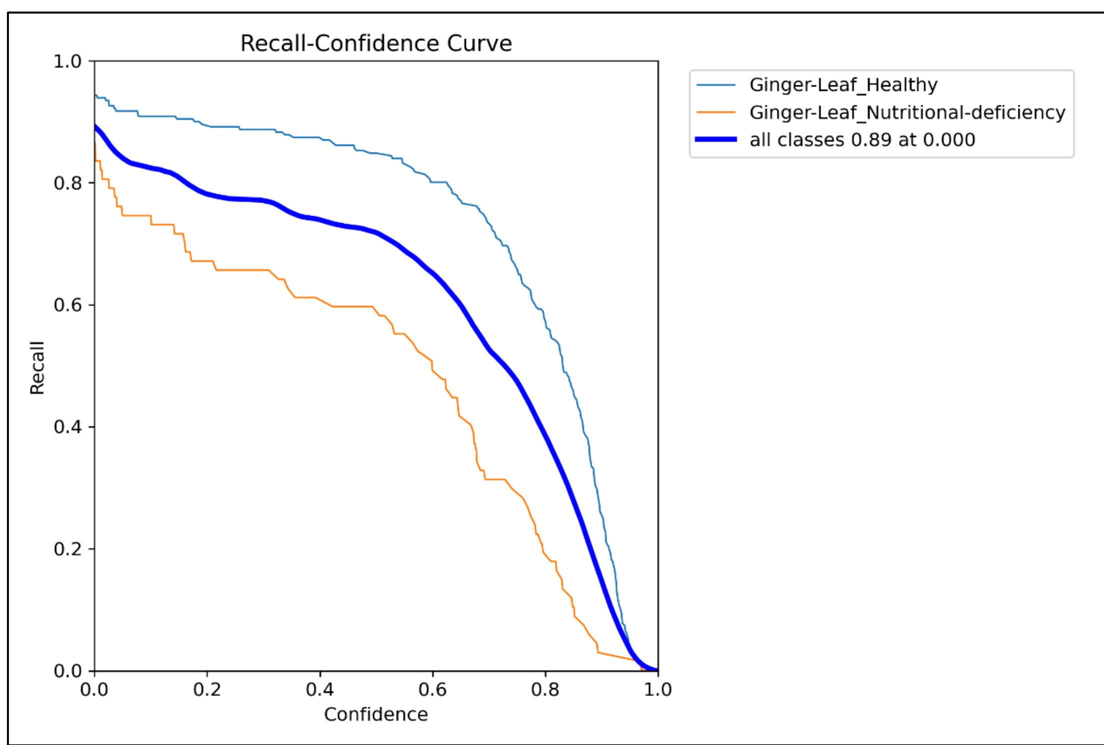



Figure 4.13: Recall-Confidence Curve

Based on the curve in Figure 4.13, as the confidence threshold increases, recall decreases because the model becomes more selective, potentially missing some true positives. The recall confidence analysis shows that the model achieves a recall score of 0.89 across all classes at a confidence level of 0.000, indicating that the model can identify a high proportion of actual healthy and unhealthy leaves when it makes predictions. However, if the confidence level is lower it will have a higher number of false positives.

To decide on a threshold that ensures a high detection rate of the objects of interest, selecting the confidence level when recall decreases steeply will be suitable. Because A model that predicts too many healthy leaves at low confidence may overwhelm users with false positives, leading to inefficiencies in agricultural monitoring.

## 4.4 Leaf Health Status Classification

The Table 4.3 shows an example of leaf detection and classification. The green masks indicate healthy leaves, while the red masks represent unhealthy leaves. The confusion matrix is shown in Table 4.4.

Table 4.3: Example Images of Detected and Classified Leaves on a Ginger Plant.

| No. | Detected Image | Plotted Image |
|---|---|---|
| 1 | | |
| 2 | | |

Table 4.4: Leaf Health Status Confusion matrix

| | Actual Healthy Leaves | Actual Unhealthy Leaves |
|---|---|---|
| Predicted Healthy Leaves | 203 | 9 |
| Predicted Unhealthy Leaves | 7 | 49 |

Based on the Table 4.4, metrics such as precision, recall, and F1-score for the leaf classification model were calculated. The results of these metrics are summarized in the Table 4.5.

Table 4.5: Leaf Health Status Confusion matrix

| | Healthy Leaves | Unhealthy Leaves |
|---|---|---|
| Accuracy | 94.03 % | |
| Precision | 95.75 % | 87.50 % |
| Recall | 96.67 % | 84.48 % |
| F1- Score | 96.21 % | 85.97 % |

From the Table 4.5, the model achieved an overall accuracy of 94.03% for detecting and classifying healthy and unhealthy leaves. However, the performance metrics for classifying unhealthy leaves, specifically precision, recall, and F1-scoreare below 90%. This indicates that the model's ability to classify unhealthy leaves is comparatively weaker.

### 4.4.1 Leaf Count Per Plant and Health Status Classification

After detecting the leaves of each ginger plant using the trained YOLOv8 model, the system counts the total number of leaves on each plant. The system uses a threshold of 50% to classify the plant as healthy or unhealthy. If more than 50% of the leaves on a plant are classified as unhealthy, the entire plant is marked as unhealthy. The formula used to classify the plant's health is:

$$\text{Health Status} = \begin{cases} \text{Unhealthy}, & \dfrac{\text{Unhealthy Leaves}}{\text{Total Leaves}} > 0.5 \\ \text{Healthy}, & \text{otherwise} \end{cases} \tag{4.1}$$

During testing, it was observed that most ginger plants with clear visual signs of disease had more than 60% unhealthy leaves, validating the threshold set for classification. For example, in one test case, a ginger plant with 7 leaves, 5 of which were classified as unhealthy, was accurately classified as unhealthy by the system.

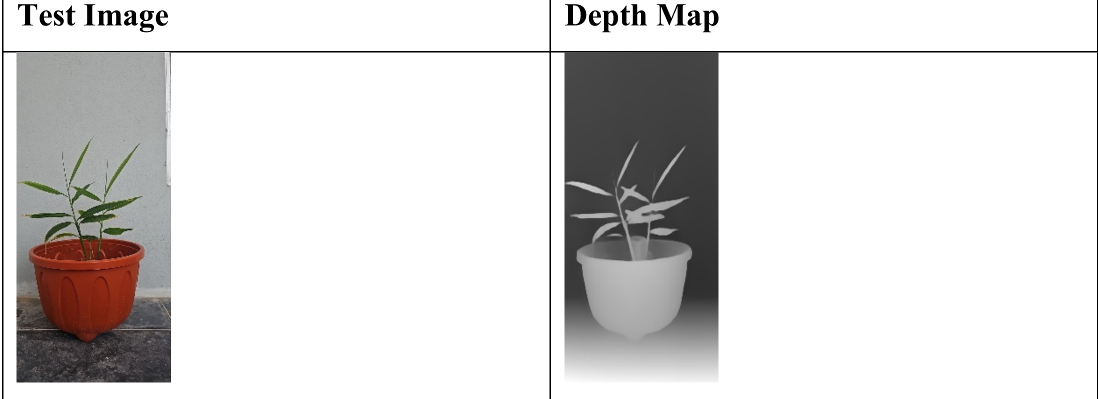
## 4.5 Depth Estimation Model

The depth estimation model, Intel’s dpt-large, was deployed to calculate the distance between the camera and the detected ginger plant. This model provides a depth map for each image, which was used to determine the distance of the target from the camera. This distance estimated is used for calculation of the plant's height and the area of its leaves.

### 4.5.1 Distance Measurement Results

The Table 4.6 shown an example of the depth map generated from the test image. The colour gradient indicates different depths, with brighter colours representing closer distances and darker colours representing farther distance

Table 4.6: Example Depth Map of a Ginger Plant

| **Test Image** | **Depth Map** |
|---|---|
|  | |

Total 15 test images with different distance are tested. The distance measurements for plants image with different distance are summarized in Table 4.3 and Figure 4.14. The distances were calculated from the depth map, providing how far each plant is from the camera.

Table 4.7: Depth Estimation Result of Test Images

| **Actual distance,** $y_i$ **(cm)** | **Predicted distance,** $\hat{y}_i$ **(cm)** | **Deviation (%)** |
|---|---|---|
| 50 | 48.60 | 2.81 |
| 60 | 60.83 | 1.38 |
| 70 | 72.23 | 3.18 |
| 80 | 82.45 | 3.06 |
| 90 | 88.20 | 2.00 |
| 100 | 104.38 | 4.38 |
| 110 | 110.12 | 0.11 |
| 120 | 113.05 | 5.79 |
| 130 | 121.52 | 6.52 |
| 140 | 130.64 | 6.69 |
| 150 | 135.23 | 9.85 |
| 160 | 137.04 | 14.35 |
| 170 | 131.89 | 22.42 |
| 180 | 137.43 | 23.65 |
| 190 | 142.44 | 25.03 |

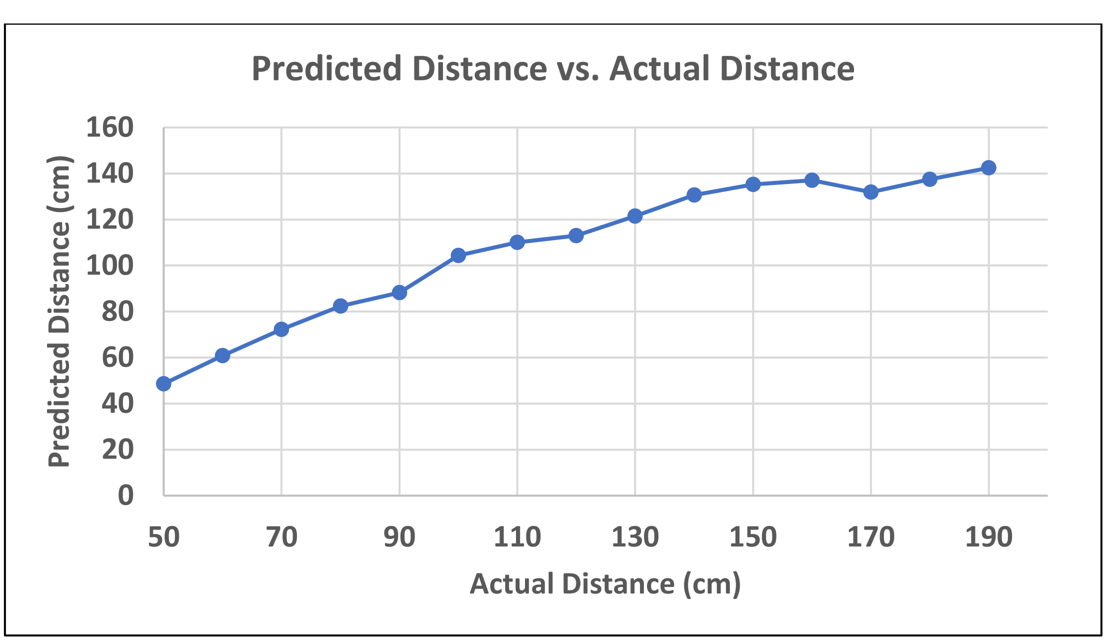



Figure 4.14: Graph of Predicted Distance vs. Actual Distance

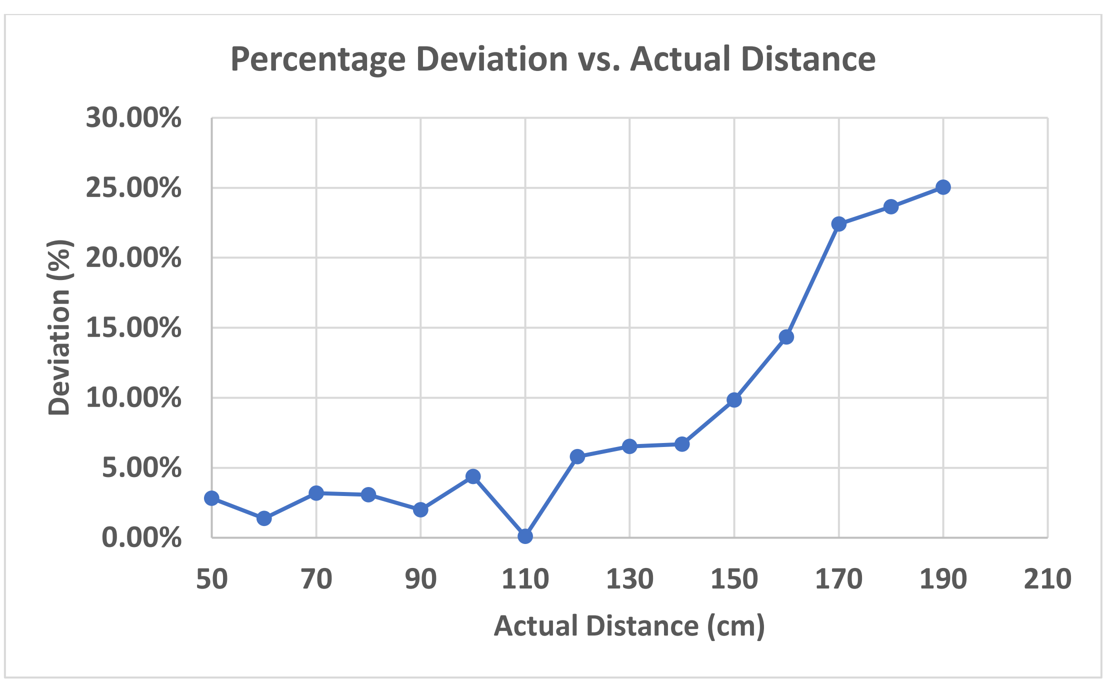



Figure 4.15: Graph of Percentage Deviation vs. Actual Distance

### 4.5.2 Model Performance

From the Figure 4.15, it is observed that for distances up to 110 cm, the deviation between the calculated and actual distance is relatively low, with the deviation percentage remaining below 10%. However, beyond 110 cm, the deviation increases significantly, reaching as high as 25.03% at 190 cm. This indicates that the model performs better at shorter distances but struggles with accuracy as the distance between the plant and the camera increases.

The depth estimation model's accuracy was further evaluated by comparing the predicted depths to ground truth measurements obtained through physical measurement techniques. The Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) were calculated to quantify the difference between the predicted and actual depths.

The Mean Absolute Error (MAE) is calculated as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4.2}$$

The Root Mean Squared Error (RMSE) is given by

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{4.3}$$

where

$y_i$ = actual depth

$\hat{y}_i$ = predicted depth

$n$ = number of samples

For the depth estimation model, based on total 15 images with different distance tested, an MAE of 13.60 cm and an RMSE of 20.84 cm were recorded, indicating that there is a slight overestimation in the predicted depths.

The depth predictions were compared to ground truth measurements using statistical analysis. The correlation coefficient $r$ between the predicted and actual depths was calculated as:

$$r = \frac{\sum\left((y_i - \bar{y})(\hat{y}_i - \bar{\hat{y}})\right)}{\sqrt{\sum(y_i - \bar{y})^2}\sqrt{\sum(\hat{y}_i - \bar{\hat{y}})^2}} \tag{4.4}$$

where

$y_i$ = actual depth

$\hat{y}_i$ = predicted depth

$\bar{y}$ = mean actual depth

$\bar{\hat{y}}$ = mean predicted depth

The correlation coefficient was found to be 0.96 which is around 96%, indicating a strong positive correlation between the model's predictions and the actual measurements.

## 4.6 Plant Height and Leaf Area Calculations

The calculated distance between the camera and target is used to calculate the height of the ginger plant and the area of each leaf. Before calculation of plant height and leaf area, the pixel height and width of the image is calculated. The pixel height and width of an image captured by the camera can be calculated using the field of view (FoV) and the image's dimensions. The Field of View (FoV) is calculated as:

$$FoV_{height} = 2 \cdot \tan^{-1}\left(\frac{S_h}{2f}\right) \tag{4.5}$$

$$FoV_{width} = 2 \cdot \tan^{-1}\left(\frac{S_w}{2f}\right) \tag{4.6}$$

where

$FoV_{height}$ = vertical field of view, radians

$FoV_{width}$ = horizontal field of view, radians

$S_h$ = sensor height of the camera, mm

$f$ = focal length of the camera, mm

The pixel height and pixel width of the image are calculated as:

$$p_{i,height} = \tan\left(\frac{FoV_{height}}{2}\right) \cdot \frac{D}{Image\ Height} \tag{4.7}$$

$$p_{i,width} = \tan\left(\frac{FoV_{width}}{2}\right) \cdot \frac{D}{Image\ Width} \tag{4.8}$$

where

$p_{i,height}$ = pixel height corresponding to the image, cm

$p_{i,width}$ = pixel width corresponding to the image, cm
$Image\ Width$ = width of the image, pixels
$Image\ Height$ = height of the image, pixels
$D$ = distance from the camera to the object, cm

By using these formulas discussed, the actual height of any object in the image plane can be determined. Therefore, the formula to calculate the actual height of the ginger plant is provided below. The observed height of the target in pixels is extracted from the plant detection model discussed in 4.2, which uses the top and the bottom coordinates of the bounding box generated from the plant detection model.

$$H = h \times p_{i,height} \tag{4.9}$$

where
$H$ = actual height of target, cm
$h$ = observed height of the target, pixels

As the total area covered by a pixel in the real world can be derived from the product of the pixel height and pixel width, the model discussed in Section 4.3, can segment the leaf from the provided image. Therefore, the segmented leaf can be used to calculate the leaf area, which represents the observed area of the target in pixels $a$. For leaf area estimation, the area $A$ was calculated using the following formula:

$$A = a \times (p_{i,width} \cdot p_{i,height}) \tag{4.10}$$

where
$A$ = actual area of target, $cm^2$
$a$ = observed area of the target, pixels

### 4.6.1 Example Calculation of Plant Height and Leaf Area

By reusing the images used to detect the distance between the target and the camera, as described in section 4.5.1, the calculated and actual plant heights are compared in the table below.

Table 4.8: Comparison Between Calculated and Actual Plant Height

| **Actual distance (cm)** | **Predicted distance (cm)** | **Actual Height (cm)** | **Calculated height (cm)** | **Deviation (%)** |
|---|---|---|---|---|
| 50 | 48.60 | 22.00 | 21.34 | 3.00 |
| 60 | 60.83 | 22.00 | 25.42 | 15.56 |
| 70 | 72.23 | 22.00 | 25.03 | 13.78 |
| 80 | 82.45 | 22.00 | 23.33 | 6.04 |
| 90 | 88.20 | 22.00 | 22.45 | 2.02 |
| 100 | 104.38 | 22.00 | 22.78 | 3.56 |
| 110 | 110.12 | 22.00 | 22.04 | 0.19 |
| 120 | 113.05 | 22.00 | 19.59 | 10.96 |
| 130 | 121.52 | 22.00 | 19.17 | 12.86 |
| 140 | 130.64 | 22.00 | 19.87 | 9.70 |
| 150 | 135.23 | 22.00 | 17.49 | 20.52 |
| 160 | 137.04 | 22.00 | 17.37 | 21.06 |
| 170 | 131.89 | 22.00 | 14.19 | 35.50 |
| 180 | 137.43 | 22.00 | 14.36 | 34.73 |
| 190 | 142.44 | 22.00 | 14.22 | 35.37 |

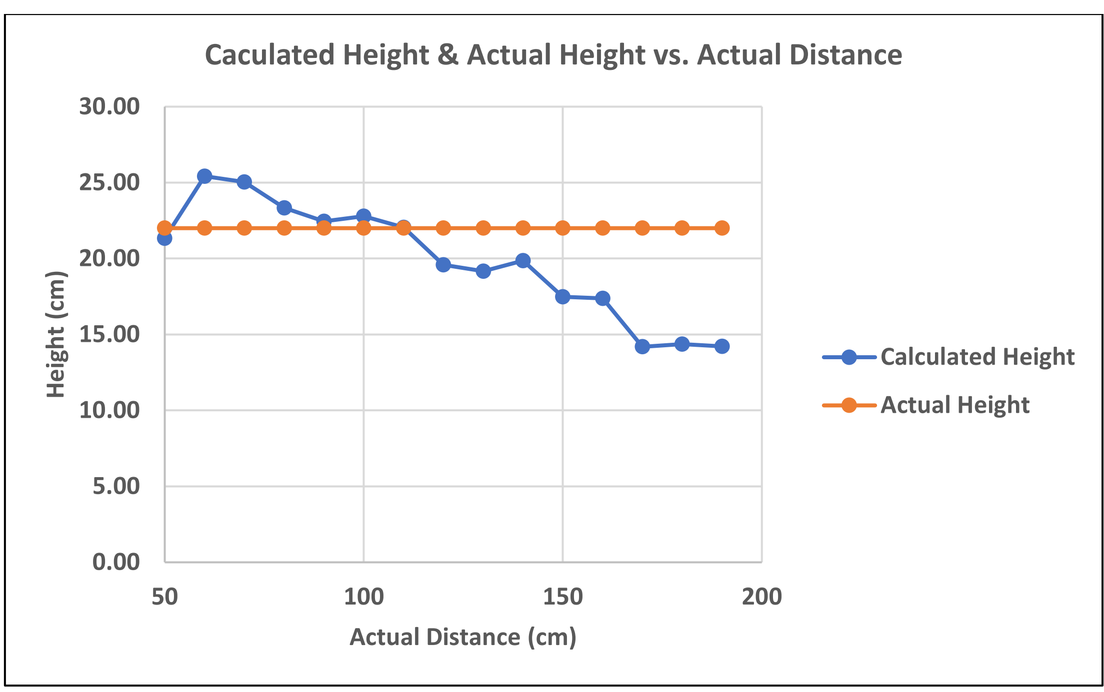



Figure 4.16: Graph of Calculated Height and Actual Height vs. Actual Distance

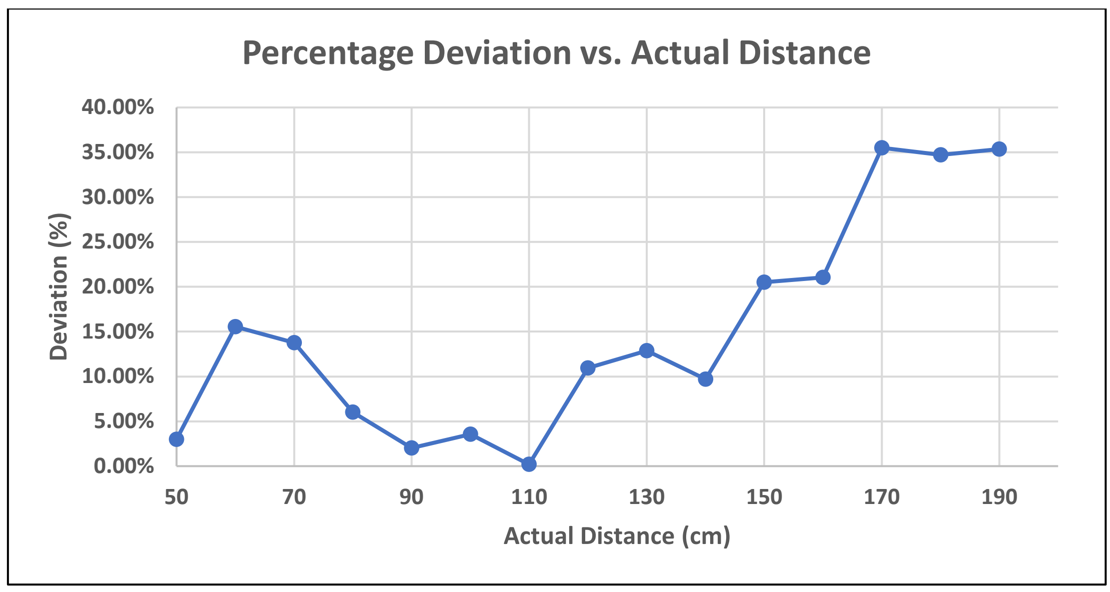



Figure 4.17: Graph of Percentage Deviation vs. Actual Distance

From the Figure 4.17, it is observed that when distances greater than 150 cm, the percentage deviation is greater than 20%. This indicates that the model performs better at shorter distances but struggles with accuracy as the distance between the plant and the camera increases.

The area of the leaves was calculated using the segmentation masks provided by the YOLOv8 model. However, unlike plant height, leaf area cannot be directly compared with an actual measurement in the field, as it is not feasible to manually measure the surface area of leaves in the same manner. Therefore, the calculated leaf area is served as relative measures to comparing the sizes of different leaves within the dataset or monitoring changes in leaf size over time. For example, the calculated leave can be use as part of disease progression or growth tracking.

## 4.7 Challenges and Limitation

One of the challenges encountered was the model's difficulty in detecting the plants and its leaves that were partially blocked by other objects. Additionally, the model shown a decrease in accuracy for far objects, as the disparity between the camera's focal length and the depth made it challenging for the model to estimate accurately. This limitation suggests that the model may require further fine-tuning or the integration of additional sensors such as LIDAR for more reliable depth estimation.

# CHAPTER 5

# CONCLUSION AND RECOMMENDATIONS

## 5.1 Conclusion

In this project, the objectives were met through the developed of integrated system combining image processing algorithms and deep learning techniques to monitor growth stage of ginger plants. A YOLOv8-based model was designed and trained to detect and classify ginger plants. Additionally, another YOLOv8 model was developed to detect and classify ginger leaves based on their health status. The addition of a depth estimation model was used for calculation of plant height and leaf area

The YOLOv8 models trained for this task has shown high accuracy and efficiency, making it suitable for real-time agricultural monitoring applications. Although there are minor differences between theoretical calculations and simulation results due to factors such as environmental noise and sensor limitations, but the system still proven to be a reliable tool for plant monitoring. The depth estimation model also showed there is need of some refinements in accuracy to further improve the overall system.

Overall, the system developed in this project shown its capability in ginger plant monitoring through advanced deep learning techniques, fulfilling the project’s goal.

## 5.2 Recommendation

Several areas can be enhanced to increase the system's performance and applicability. Firstly, the accuracy of the depth estimation model could be enhanced by using a hardware sensor such as a LIDAR sensor to enhance the estimate of plant height along with the area of the leaf.

Moreover, the difference in theoretical value and simulation value with the trained YOLOv8 models and depth estimation model was due to insufficient datasets. Therefore, the research could be extended to increase the complexity and size of the training dataset, which may enhance the model’s applicability to distinct conditions. In addition, it could be conjectured that incorporating additional data from other environmental conditions, plant stages, or other types of sensors, would help increase performance on the training data.

The system should be applied to real-world farming practices for testing of other factors within the agricultural setting including but not limited to changes in lighting and occlusion of plants. Besides, the future developments

# REFERENCES


Al-amri, S. S., Kalyankar, N. V. & D, K. S., 2010. Image Segmentation by Using Threshold Techniques', Journal of Computing. *Journal of Computing,* 2(5), p. 83–86.

C, S., JaganMohan, K. & Arulaalan, M., 2022. Real Time Riped Fruit Detection using Faster R-CNN Deep Neural Network Models. *2022 International Conference on Smart Technologies and Systems for Next Generation Computing (ICSTSN),* 25 3.p. 1–4.

Erhan, D., Szegedy, C., Toshev, A. & Anguelov, D., 2013. Scalable Object Detection using Deep Neural Networks. 8 12.

FONG, C. O., 1990. SMALL AND MEDIUM INDUSTRIES IN MALAYSIA: ECONOMIC EFFICIENCY AND ENTREPRENEURSHIP. *The Developing Economies,* 6, 28(2), pp. 152-179.

Ganesan, P., Rajini, V., Rajini, B. & Basha.Shaik, K., 2014. HSV Color Space Based Segmentation of Region of Interest in Satellite Images. *International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT),* pp. 101-105.

Hussain, M., 2023. YOLO-v1 to YOLO-v8, the Rise of YOLO and Its Complementary Nature toward Digital Manufacturing and ndustrial Defect Detection. *Machines,* Volume 11, p. 677.

Lee, S. H., Chan, C. S., Mayo, S. J. & Remagnino, P., 2017. How deep learning extracts and learns leaf features for plant classification. *Pattern Recognition,* 11, Volume 71, pp. 1-13.

Lee, U. et al., 2018. An automated, high-throughput plant phenotyping system using machine learning-based plant segmentation and image analysis. 27 4, 13(4), p. 0196615.

Lin, T.-Y.et al., 2017. Feature Pyramid Networks for Object Detection. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* pp. 936-944.

Lou, H. et al., 2023. DC-YOLOv8: Small-Size Object Detection Algorithm Based on Camera Sensor. *Electronics,* 21 5, 12(10), p. 2323.

Manjula, D. V., 2017. Image Edge Detection and Segmentation by using Histogram Thresholding method. *International Journal of Engineering Research and Applications,* 8, 07(08), pp. 10-16.

Matahir, H. & Tuyon, J., 2013. The Dynamic Synergies between Agriculture Output and Economic Growth in Malaysia. *International Journal of Economics and Finance,* 18 3.5(4).

Pathak, A. R., Pandey, M. & Rautaray, S., 2018. Application of Deep Learning for Object Detection. *Procedia Computer Science,* Volume 132, pp. 1706-1717.

Porebski, A., Vandenbroucke, N. & Macaire, L., 2008. Haralick feature extraction from LBP images for color texture classification. *First Workshops on Image Processing Theory, Tools and Applications,* pp. 1-8.

Praveen, J. K. & Domnic, . S., 2019. Image based leaf segmentation and counting in rosette plants. *Information Processing in Agriculture,* 6, 6(2), pp. 233-246.

R S, Y., Shamsher, P. & K., S., 2024. A YOLOv8-based Model for Precise Corrosion Segmentation in Industrial Imagery.

Rashid, M. N. & Fadzil, L. M., 2023. Comparative Review of Object Detection Algorithms in Small Single-Board Computers. *International Journal on Recent and Innovation Trends in Computing and Communication,* 1 9, 11(7), p. 244–252.

Redmon, J., Divvala, S., Girshick, R. & Farhadi, A., 2016. You Only Look Once: Unified, Real-Time Object Detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* Volume 779-788.

Ren, S., He, K., Girshick, R. & Sun, J., 2017. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 1 6, 39(6), pp. 1137-1149.

Reswara , E., Suakanto , S. & Putra , S. A., 2023. Comparison of Object Detection Algorithm using YOLO vs Faster R-CNN : A Systematic Literature Review. *ICBDT '23: Proceedings of the 2023 6th International Conference on Big Data Technologies,* 9.pp. 419-424.

Salman, N., 2006. Image Segmentation Based on Watershed and Edge Detection Techniques. *The International Arab Journal of Information Technology,* 4, 3(2), pp. 104-110.

Sandro Luis de, A. et al., 2024. Segmentação de Pólipos em Imagens de Colonoscopia utilizando YOLOv8.

Selvaraj, M. G. et al., 2019. AI-powered banana diseases and pest detection. *Plant Methods,* 12 8.15(1).

Shrestha, A. & Mahmood, A., 2019. Review of Deep Learning Algorithms and Architectures. *IEEE,* 22 4, Volume 7, pp. 53040 - 53065.

Terven, J., Córdova-Esparza, D.-M. & Romero-González, J.-A., 2024. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Mach. Learn. Knowl. Extr.,* 5(4), pp. 1680-1716.

Tong, Y.-S., Lee, T.-H. & Yen, K.-S., 2022. Deep Learning for Image-Based Plant Growth Monitoring: A Review. *International Journal of Engineering and Technology Innovation,* 5, 12(3), pp. 225-246.

Wang, C.-Y.et al., 2020. CSPNet: A New Backbone that can Enhance Learning Capability of CNN. *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW),* pp. 1571-1580.

Wang, P. et al., 2024. Leaf Segmentation Using Modified YOLOv8-Seg Models. *Reproductive and developmental Biology,* 14(6), p. 780.

Wu, Y. et al., 2024. YOLOv8-segANDcal: segmentation, extraction, and calculation of soybean radicle features. *Frontiers in Plant Science,* Volume 15.

Zineb, S., Lyamine, G., Kamel, B. & Rezki, A., 2023. IoV Data Processing Algorithms for Automatic Real-Time Object Detection - Literature Review. *2023 International Conference on Inventive Computation Technologies (ICICT),* 26 4.pp. 1335-1342.

# APPENDICES

## APPENDIX A: Code for Training Yolo Model in Google Colab

```
!pip install ultralytics
from IPython.display import clear_output
clear_output()

import shutil
import os
import torch
import ultralytics
from ultralytics import YOLO
from IPython.display import clear_output

clear_output()
ultralytics.checks()

%cd /content
HOME = os.getcwd()

Dataset_path = "/content/drive/MyDrive/FYP/Datasets/leaf-
detection.v12i.yolov8" #@param {type:"string"}

# Ensure the destination folder exists
destination_folder = os.path.join(HOME,
"datasets",os.path.basename(Dataset_path))
os.makedirs(destination_folder, exist_ok=True)

# Copy the entire folder
try:
  print(f"Copying folder from {Dataset_path} to
{destination_folder}")
  shutil.copytree(Dataset_path,
                  destination_folder,
                  dirs_exist_ok=True)
  print("Folder copied successfully!")
```

```
except Exception as e:
  raise ValueError(f"Error copying folder: {e}")

Data_yaml_path = os.path.join(Dataset_path, "data.yaml")

# Create YOLOv8 model
Model_path = "yolov8n-seg.pt" #@param {type:"string"}
try:
  # load a pretrained model (recommended for training)
  model = YOLO(Model_path)

except Exception as e:
  raise ValueError("Model path is invalid >> "+str(Model_path))

# Check if GPU is available
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"Using device: {device}")

# Move model to GPU
model.to(device)

# Start training
model.train(data=Data_yaml_path, epochs=40, device=device)
print("Training completed!")

# Define source and destination paths
source_folder = "/content/runs"
base_destination_folder =
os.path.join("/content/drive/MyDrive/export_runs", "runs")

# Function to get a unique destination folder path
def get_unique_destination_folder(base_folder):
    suffix = 1
    destination_folder = f"{base_folder}{suffix}"

    while os.path.exists(destination_folder):
        suffix += 1
        destination_folder = f"{base_folder}{suffix}"

    return destination_folder

# Get a unique destination folder path
destination_folder =
get_unique_destination_folder(base_destination_folder)

# Copy the folder
try:
    print(f"Copying folder from {source_folder} to
{destination_folder}")
```

```
    os.makedirs(destination_folder, exist_ok=True)
    shutil.copytree(source_folder, destination_folder,
dirs_exist_ok=True)
    print("Folder copied successfully!")
except Exception as e:
    raise ValueError(f"Error copying folder: {e}")
```

APPENDIX B: Code for Real-Time Monitoring Interface in Python Language

```
import sys
import os
import time
import json
import pickle
import re
import natsort
from collections import Counter
from queue import Queue
import cv2
import numpy as np
from collections import defaultdict
from functools import partial
from TargetDetection4 import *
from PyQt5.QtWidgets import (
    QApplication, QMainWindow, QAction, QHBoxLayout, QVBoxLayout,
QLabel, QFrame,
    QScrollArea, QWidget, QComboBox, QPushButton, QFormLayout,
    QSizePolicy, QFileDialog, QLineEdit, QTabWidget)
from PyQt5.QtCore import Qt, QTimer, QMargins, pyqtSlot, QThread,
pyqtSignal, QMutex, QMutexLocker
from PyQt5.QtChart import QChart, QPieSeries, QPieSlice, QChartView
from PyQt5.QtGui import QIcon, QColor, QPainter
import qdarktheme

# Default to webcam. Can be changed to a video file path or image
path.
# input_source =
r"C:\Users\yougt\Documents\Python\fyp\code\YOLO_venv\Images\ginger
plant video\20240719-015654.mp4"
# input_source =
r"C:\Users\yougt\Documents\Python\fyp\code\YOLO_venv\cropped"
```

```
# input_source = 0

light_qss = """
QFrame {
    border-width: 2px;
    border-color: black;
}
QLabel {
    font-size: 10pt;
}
QPushButton {
    font-size: 10pt;
}
QComboBox {
    font-size: 10pt;
}
QTabWidget {
    font-size: 10pt;
}
QChart {
    font-size: 10pt;
}
QPieSlice {
    font-size: 10pt;
}
QPieSeries {
    font-size: 10pt;
}
QFont {
    font-size: 10pt;
}
QLineEdit {
    font-size: 10pt;
}
"""

dark_qss = """
QFrame {
    border-width: 2px;
    border-color: light;
}
QLabel {
    font-size: 10pt;
}
QPushButton {
    font-size: 10pt;
}
QComboBox {
    font-size: 10pt;
```

```
}
QTabWidget {
    font-size: 10pt;
}
QChart {
    font-size: 10pt;
}
QPieSlice {
    font-size: 10pt;
}
QPieSeries {
    font-size: 10pt;
}
QFont {
    font-size: 10pt;
}
QLineEdit {
    font-size: 10pt;
}
"""

class RealTimeVideoApp(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Real-Time Plant Monitoring")

        # Load setting & Initialize target detection model and tools
        settings = self.load_settings()
        self.distance_scale = settings['distance_scale']
        self.sensor_height = settings['sensor_height']
        self.sensor_width = settings['sensor_width']
        self.focal_length = settings['focal_length']
        self.set_theme(settings['theme'])
        self.model_plant_path = settings['model_plant_path']
        self.model_leaf_path = settings['model_leaf_path']
        self.TargetDetection = TargetDetect(self.model_plant_path,
self.model_leaf_path)
        self.set_verbose(settings['verbose'])

        # Initialize window
        self.setGeometry(100, 100, 1700, 900)
        self.create_menu_bar()

        self.central_layout = QHBoxLayout()
        central_widget = QWidget(self)
        central_widget.setLayout(self.central_layout)
        self.setCentralWidget(central_widget)
```

```
        self.DetectTarget = DetectTarget(self.TargetDetection,
self.verbose)
        self.DetectLeaf = DetectLeaf(self.TargetDetection,
self.verbose, self.distance_scale, self.sensor_height,
self.sensor_width, self.focal_length)

        # self.start_processing(input_source)

    def select_input(self, type_of_source):
        if type_of_source == "dir":
            # Create a file dialog and get the selected file path
            input_source = QFileDialog.getExistingDirectory(self,
"Select a Folder")
        else:
            # Create a file dialog and get the selected file path
            options = QFileDialog.Options()
            input_source, _ = QFileDialog.getOpenFileName(self,
"Select a File", "", ";All Files (*)", options=options)

        if input_source:
            if (self.DetectTarget and self.DetectTarget.isRunning()):
                print('Waiting')
                self.DetectTarget.wait()  # Wait until the thread
finishes

            if (self.DetectLeaf and self.DetectLeaf.isRunning()):
                print('Waiting')
                self.DetectLeaf.wait()  # Wait until the thread
finishes

            if hasattr(self, "VideoProcessor"):
                del self.VideoProcessor

            if hasattr(self, "DirectoryProcessor"):
                del self.DirectoryProcessor

            self.start_processing(input_source)

    def select_webcam(self):
        if hasattr(self, "VideoProcessor"):
                del self.VideoProcessor
        if hasattr(self, "DirectoryProcessor"):
            del self.DirectoryProcessor

        self.start_processing(0)

    def start_processing(self, input_source):
        # Check if the input source is a valid video file or webcam.
        if input_source == 0 or (os.path.exists(input_source) and
```

```
                                   os.path.isfile(input_source) and
                                   input_source.lower().endswith(('.m
p4', '.avi', '.mov', '.mkv'))):

            self.VideoProcessor = VideoProcessor(self, input_source,
self.DetectTarget, self.DetectLeaf)
            self.DetectLeaf.set_rest(True)
            # Setup video  timers
            self.timer_video = QTimer()
            self.timer_video.timeout.connect(self.VideoProcessor.run)

            self.set_layout_to_central_widget("main")

        # Check if the input source is a valid image file.
        elif (os.path.exists(input_source) and
              os.path.isfile(input_source) and
              input_source.lower().endswith(('.png', '.jpg',
'.jpeg'))):
            k=0

        # Check if the input source is a directory.
        elif (os.path.exists(input_source) and
              os.path.isdir(input_source)):

            self.DirectoryProcessor = DirectoryProcessor(self,
input_source, self.DetectTarget, self.DetectLeaf)

            # Setup image timer
            self.timer_image = QTimer()
            self.timer_image.timeout.connect(self.show_next_image)

            self.set_layout_to_central_widget("main")

            self.DirectoryProcessor.run()

        else:
            raise ValueError("Unsupported input source")

    def pause_timer(self, timer, pause_button, interval):
        timer.stop()
        pause_button.setText("Play")
        pause_button.clicked.connect(lambda :self.start_timer(
            timer, pause_button, interval))

    def start_timer(self, timer, pause_button, interval):
        timer.start(interval)
        pause_button.setText("Pause")
        pause_button.clicked.connect(lambda :self.pause_timer(
            timer, pause_button, interval))
```

```
    def show_previous_image(self):
        self.DirectoryProcessor.show_previous_image()

    def show_next_image(self):
        self.DirectoryProcessor.show_next_image()

    def update_image(self, image = None):
        if image is not None:
            self.current_image = image
        if hasattr(self,"main_layout") and self.main_layout:
            if hasattr(self,"current_image"):
                if isinstance(self.current_image, str):
                    self.image_container.setText(image)
                else:
                    pixmap = preprocess_input(self.current_image,
self.image_container.width()-2, self.image_container.height()-2)
                    self.image_container.setPixmap(pixmap)

    def update_image_detail(self, text):
        if self.main_layout:
            self.image_detail_label.setText(text)

    def update_gallery(self, plant_datas):
        # Check if main_layout exist
        if self.main_layout:
            # Save current scroll position
            scroll_position_1 =
self.scroll_area_recognized.verticalScrollBar().value()
            scroll_position_2 =
self.scroll_area_unrecognized.verticalScrollBar().value()

            # Clear scrollable area
            self.clear_layout_and_widget(self.content_layout_recogniz
ed)
            self.clear_layout_and_widget(self.content_layout_unrecogn
ized)

            for plant_details in plant_datas:
                # Create gallery container
                plant_image_label, gallery_container,
plant_detail_container, details_button  =
self.create_gallary_container()

                # Preprocess input image
                pixmap =
preprocess_input(plant_details['plant_image'], 250, 250)

                # insert preproccessed image to plant_image_label
```

```
                plant_image_label.setPixmap(pixmap)

                # Connect signals to slot
                details_button.clicked.connect(partial(self.show_imag
e_details, plant_details))

                details = {'Name': plant_details['plant_label'],
                        'Height':
f"{plant_details['plant_height']:.4f} cm",
                        'Distance':
f"{plant_details['plant_distance']:.4f} cm",
                        'Leaf count': plant_details['leaf_count'],
                        'Disease': plant_details['plant_disease']
                        }

                # Loop through the data and create QLabel widgets
                for title, detail in details.items():
                    detail_label = QLabel(f"<b>{title}:</b>
{detail}")
                    detail_label.setFixedWidth(300)  # Set the
maximum width for the label
                    detail_label.setWordWrap(True)  # Enable word
wrapping
                    plant_detail_container.addWidget(detail_label)

                # Add gallery_container to content_Layout
                if plant_details['plant_label'] == "Ginger":
                    self.content_layout_recognized.addLayout(gallery_
container)
                else:
                    self.content_layout_unrecognized.addLayout(galler
y_container)

            # Restore the scroll position
            self.scroll_area_recognized.verticalScrollBar().setValue(
scroll_position_1)
            self.scroll_area_unrecognized.verticalScrollBar().setValu
e(scroll_position_2)

    def show_image_details(self, details):
        '''Create and show the new top-level window'''
        self.top_window = TopWindow(details)
        self.top_window.show()

    def set_layout_to_central_widget(self, layout):
        # Clear the current layout and show settings view
        self.clear_layout_and_widget(self.central_layout)

        if layout == "main":
```

```
            (self.main_layout,
             self.image_detail_label, self.image_container,
option_container,
             self.scroll_area_recognized,
self.content_layout_recognized,
             self.scroll_area_unrecognized,
self.content_layout_unrecognized) = self.create_main_layout()

            if hasattr(self, "VideoProcessor") and
self.VideoProcessor:
                pause_button = QPushButton("Start")
                pause_button.clicked.connect(lambda :self.start_timer
(
                    self.timer_video, pause_button, 70))

                option_container.addWidget(pause_button)
                pause_button.click()

            elif hasattr(self, "DirectoryProcessor") and
self.DirectoryProcessor:
                prev_button = QPushButton("Previous")
                auto_button = QPushButton("Play")
                next_button = QPushButton("Next")
                option_container.addWidget(prev_button)
                option_container.addWidget(auto_button)
                option_container.addWidget(next_button)
                # Connect signals to slots
                prev_button.clicked.connect(lambda:
self.show_previous_image())
                auto_button.clicked.connect(lambda :self.start_timer(
                    self.timer_image, auto_button, 3000))
                next_button.clicked.connect(lambda:
self.show_next_image())

            self.central_layout.addLayout(self.main_layout)

        elif layout == "setting":
            if hasattr(self,"timer_image"):
                self.timer_image.stop()
            if hasattr(self,"timer_video"):
                self.timer_video.stop()

            self.setting_layout = self.create_setting_display()
            self.central_layout.addLayout(self.setting_layout)

    def create_menu_bar(self):
        # Create the menu bar
        menu_bar = self.menuBar()
```

```
        # File menu
        file_menu = menu_bar.addMenu("File")
        settings_action = QAction("Preference", self)
        nothing_action = QAction("Nothing", self)
        exit_action = QAction("Exit", self)
        exit_action.triggered.connect(self.close)  # Connect the exit
action to close the app

        file_menu.addAction(settings_action)
        file_menu.addAction(nothing_action)
        file_menu.addSeparator()  # Add a separator line
        file_menu.addAction(exit_action)

        # Add functionality to the actions
        settings_action.triggered.connect(lambda:
self.set_layout_to_central_widget("setting"))

        select_input_menu = menu_bar.addMenu("Select Input")
        select_dir_action = QAction("Select Folder", self)
        select_file_action = QAction("Select File", self)
        select_webcam_action = QAction("Select Webcam", self)

        select_input_menu.addAction(select_dir_action)
        select_input_menu.addAction(select_file_action)
        select_input_menu.addAction(select_webcam_action)

        # Add functionality to the actions
        select_dir_action.triggered.connect(lambda:
self.select_input("dir"))
        select_file_action.triggered.connect(lambda:
self.select_input("file"))
        select_webcam_action.triggered.connect(lambda:
self.select_webcam())

    def create_main_layout(self):
        realtime_display_layout = QHBoxLayout()

        # Set up image display area
        image_layout = QVBoxLayout()

        image_detail_label = QLabel("")
        image_detail_label.setFrameStyle(QFrame.Box | QFrame.Plain)
        image_detail_label.setAlignment(Qt.AlignCenter)
        image_detail_label.setFixedHeight(60)

        image_label = QLabel("Video Display Area")
        image_label.setFrameStyle(QFrame.Box | QFrame.Plain)
        image_label.setAlignment(Qt.AlignCenter)
        image_label.setMinimumSize(800, 200)
```

```
        option_frame = QFrame()
        option_frame.setFixedHeight(60)
        option_frame.setFrameStyle(QFrame.Box | QFrame.Plain)
        option_layout = QHBoxLayout()
        option_frame.setLayout(option_layout)
        option_layout.setAlignment(Qt.AlignCenter)

        # Add widget to video_layout
        image_layout.addWidget(image_detail_label)
        image_layout.addWidget(image_label)
        image_layout.addWidget(option_frame)

        # Create tabs
        tab_layout, tab_widget = self.create_tabs()
        scroll_area_recognized, content_Layout_recognized =
self.create_scroll_area()
        scroll_area_unrecognized, content_Layout_unrecognized =
self.create_scroll_area()

         # Add tabs to the QTabWidget
        tab_names = ["Recognized Plant", "Unrecognized Plant"]
        widgets = [scroll_area_recognized, scroll_area_unrecognized]
        for widget, tab_name in zip(widgets, tab_names):
            tab_widget.addTab(widget, tab_name)

        # Add widget & layout to main_layout
        realtime_display_layout.addLayout(image_layout)
        realtime_display_layout.addLayout(tab_layout)

        return (realtime_display_layout,
                image_detail_label, image_label, option_layout,
                scroll_area_recognized, content_Layout_recognized,
                scroll_area_unrecognized,
content_Layout_unrecognized)

    def create_setting_display(self):
        # Create layout for the settings window
        setting_display_layout = QFormLayout()
        setting_display_layout.setAlignment(Qt.AlignTop)

        # Create a back button with an icon
        back_button = QPushButton("Back")
        back_button.setFixedWidth(100)
        back_button.setIcon(QIcon.fromTheme("go-previous"))  # Using
a system icon

        # Create and add widgets for theme selection
        theme_label = QLabel(f"<b>Select Theme:</b>")
```

```
        theme_label.setFixedWidth(250)
        theme_combo = QComboBox()
        theme_combo.addItems(["Light", "Dark"])
        if self.theme == "light":
            theme_combo.setCurrentText("Light")
        else:
            theme_combo.setCurrentText("Dark")

        # Create and add widgets for showing detection speed
        detection_speed_label = QLabel(f"<b>Display Speed</b>")
        detection_speed_label.setFixedWidth(250)
        detection_speed_combo = QComboBox()
        detection_speed_combo.addItems(["Show", "Hidden"])
        if self.verbose:
            detection_speed_combo.setCurrentText("Show")
        else:
            detection_speed_combo.setCurrentText("Hidden")

        # Create and add widgets for adjust distance scale
        distance_scale_label = QLabel(f"<b>Distance Scale</b>")
        distance_scale_label.setFixedWidth(250)

        # Create a QLineEdit for file path
        distance_scale_textbox = QLineEdit()
        distance_scale_textbox.setText(f"{self.distance_scale}")

        # Create Horizontal layout for the row
        sensor_size_layout = QHBoxLayout()

        # Create and add widgets for adjust distance scale
        sensor_size_label = QLabel(f"<b>Sensor Size (w x h)</b>")
        sensor_size_label.setFixedWidth(250)

        # Create a QLineEdit for sensor_size
        sensor_width_textbox = QLineEdit()
        sensor_width_textbox.setText(f"{self.sensor_width}")
        multiply_label = QLabel("mm x ")
        sensor_height_textbox = QLineEdit()
        sensor_height_textbox.setText(f"{self.sensor_height}")
        unit_label = QLabel("mm")

        # Add Widget to the horizontal layout
        sensor_size_layout.addWidget(sensor_width_textbox)
        sensor_size_layout.addWidget(multiply_label)
        sensor_size_layout.addWidget(sensor_height_textbox)
        sensor_size_layout.addWidget(unit_label)

        # Create and add widgets for adjust focal_length
        focal_length_label = QLabel(f"<b>Focal Length</b>")
```

```
focal_length_label.setFixedWidth(250)

# Create a QLineEdit for file path
focal_length_textbox = QLineEdit()
focal_length_textbox.setText(f"{self.focal_length}")

# Create and add widgets for model selection
Model_selection_label = QLabel(f"<b>Model selection</b>")

# Create and add widget for Select model for plant detection
plant_model_label = QLabel(f"Plant detection")

# Create horizontal layout for the row
plant_model_layout = QHBoxLayout()

# Create a QLineEdit for file path
plant_model_file_path_textbox = QLineEdit()
plant_model_file_path_textbox.setText(self.model_plant_path)

# Create a QPushButton to open file dialog
plant_model_open_file_button = QPushButton("Open File")

# Add QLineEdit and QPushButton to the horizontal layout
plant_model_layout.addWidget(plant_model_file_path_textbox)
plant_model_layout.addWidget(plant_model_open_file_button)

# Create and add widget for Select model for leaf detection
leaf_model_label = QLabel(f"Leaf detection")

# Create horizontal layout for the row
leaf_model_layout = QHBoxLayout()

# Create a QLineEdit for file path
leaf_model_file_path_textbox = QLineEdit()
leaf_model_file_path_textbox.setText(self.model_leaf_path)

# Create a QPushButton to open file dialog
leaf_model_open_file_button = QPushButton("Open File")

# Add QLineEdit and QPushButton to the horizontal layout
leaf_model_layout.addWidget(leaf_model_file_path_textbox)
leaf_model_layout.addWidget(leaf_model_open_file_button)

# Create a save button
save_button = QPushButton("Save")
save_button.setFixedWidth(100)

# Add the everything to the form layout
setting_display_layout.addRow(back_button)
```

```
        setting_display_layout.addRow(theme_label, theme_combo)
        setting_display_layout.addRow(self.create_separator())
        setting_display_layout.addRow(detection_speed_label,
detection_speed_combo)
        setting_display_layout.addRow(self.create_separator())
        setting_display_layout.addRow(distance_scale_label,
distance_scale_textbox)
        setting_display_layout.addRow(self.create_separator())
        setting_display_layout.addRow(sensor_size_label,
sensor_size_layout)
        setting_display_layout.addRow(self.create_separator())
        setting_display_layout.addRow(focal_length_label,
focal_length_textbox)
        setting_display_layout.addRow(self.create_separator())
        setting_display_layout.addRow(Model_selection_label)
        setting_display_layout.addRow(plant_model_label,
plant_model_layout)
        setting_display_layout.addRow(leaf_model_label,
leaf_model_layout)
        setting_display_layout.addRow(self.create_separator())
        setting_display_layout.addRow(save_button)

        # Connect signals to slots
        back_button.clicked.connect(
            lambda: self.set_layout_to_central_widget("main"))
        theme_combo.currentIndexChanged.connect(
            lambda: self.set_theme(theme_combo.currentText()))
        plant_model_open_file_button.clicked.connect(
            lambda:
self.open_file_dialog(plant_model_file_path_textbox))
        leaf_model_open_file_button.clicked.connect(
            lambda:
self.open_file_dialog(leaf_model_file_path_textbox,
"model_leaf_path"))
        save_button.clicked.connect(
            lambda: self.save_settings_from_button(
                {
                    "theme": theme_combo.currentText(),
                    "verbose": detection_speed_combo.currentText(),
                    "distance_scale": distance_scale_textbox.text(),
                    "sensor_height": sensor_height_textbox.text(),
                    "sensor_width": sensor_width_textbox.text(),
                    "focal_length": focal_length_textbox.text(),
                    "model_plant_path":
plant_model_file_path_textbox.text(),
                    "model_leaf_path":
leaf_model_file_path_textbox.text(),
                }
                )
```

```
            )

        return setting_display_layout

    def create_tabs(self):
        # Create QVBoxLayout for tab_widget
        tab_layout = QVBoxLayout()

        # Create the QTabWidget
        tab_widget = QTabWidget()
        tab_widget.setFixedWidth(700)
        tab_layout.addWidget(tab_widget)

        return tab_layout, tab_widget

    def create_scroll_area(self):
        scroll_area = QScrollArea()
        scroll_area.setWidgetResizable(True)

        content_widget = QWidget()
        content_Layout = QVBoxLayout(content_widget)
        content_Layout.setAlignment(Qt.AlignTop)
        scroll_area.setWidget(content_widget)

        return scroll_area, content_Layout

    def create_separator(self):
        separator = QFrame()
        separator.setFrameShape(QFrame.HLine)
        separator.setSizePolicy(QSizePolicy.Expanding,
QSizePolicy.Minimum)
        separator.setLineWidth(3)
        return separator

    def create_gallary_container(self):
        # Create a horizontal layout for the row
        gallery_container = QHBoxLayout()

        # Create a label for the image
        plant_image_label = QLabel()
        plant_image_label.setFixedHeight(250)
        plant_image_label.setFixedWidth(250)
        plant_image_label.setFrameStyle(QFrame.Box | QFrame.Plain)
        plant_image_label.setAlignment(Qt.AlignCenter)

        # Create a detail container for the image details
        plant_detail_container = QVBoxLayout()
        plant_detail_container.setAlignment(Qt.AlignLeft |
Qt.AlignTop)
```

```
        # Create a button for details
        plant_details_button = QPushButton("Details")
        plant_details_button.setFixedWidth(90)  # Set a fixed width
for the label

        # Add widgets & layout to the gallery_container layout
        gallery_container.addWidget(plant_image_label)
        gallery_container.addLayout(plant_detail_container)
        gallery_container.addWidget(plant_details_button)

        return plant_image_label, gallery_container,
plant_detail_container, plant_details_button

    def save_settings_from_button(self, setting_dict):
        # Validate conditions before saving
        for key, value in setting_dict.items():
            if key == "model_leaf_path" or key == "model_plant_path":
                if not os.path.isfile(value) or not
value.endswith('.pt'):
                    pass

            elif key == "distance_scale" or key == "sensor_height" or
key == "sensor_width" or key == "focal_length":
                try:
                    value = float(value)
                    if key == "distance_scale":
                        self.set_sensor_size(distance_scale=value)
                    elif key == "sensor_height":
                        self.set_sensor_size(sensor_height=value)
                    elif key == "sensor_width":
                        self.set_sensor_size(sensor_width=value)
                    elif key == "focal_length":
                        self.set_sensor_size(focal_length=value)
                except:
                    pass

            elif key == "verbose":
                self.set_verbose(value)

            self.save_settings(key, value)

    def load_settings(self):
        try:
            with open("settings.json", "r") as file:
                return json.load(file)
        except FileNotFoundError:
            return {}
```

```
    def save_settings(self, key, value):
        settings = self.load_settings()
        settings[key] = value
        with open("settings.json", "w") as file:
            json.dump(settings, file, indent=4)

    def set_theme(self, selected_theme):
        if selected_theme == "Dark":
            qdarktheme.setup_theme("dark", additional_qss=dark_qss)
            self.theme  = "dark"
        else:
            qdarktheme.setup_theme("light", additional_qss=light_qss)
            self.theme = "light"

    def set_verbose(self, verbose):
        if verbose == "Show":
            self.verbose = True
            if hasattr(self, "DetectTarget"):
                self.DetectTarget.set_verbose(True)
            if hasattr(self, "DetectLeaf"):
                self.DetectLeaf.set_verbose(True)

        else:
            self.verbose = False
            if hasattr(self, "DetectTarget"):
                self.DetectTarget.set_verbose(False)
            if hasattr(self, "DetectLeaf"):
                self.DetectLeaf.set_verbose(False)

    def set_sensor_size(self, distance_scale=None,
sensor_height=None, sensor_width=None, focal_length=None):
        if distance_scale is not None:
            self.distance_scale = distance_scale
        if sensor_height is not None:
            self.sensor_height = sensor_height
        if sensor_width is not None:
            self.sensor_width = sensor_width
        if focal_length is not None:
            self.focal_length = focal_length

        if hasattr(self, "DetectLeaf"):
            self.DetectLeaf.set_sensor_size(self.distance_scale,
self.sensor_height, self.sensor_width, self.focal_length)

    def open_file_dialog(self, text_box):
        # Create a file dialog and get the selected file path
        options = QFileDialog.Options()
```

```
        file_path, _ = QFileDialog.getOpenFileName(self, "Select a
File", "", "PyTorch Model Files (*.pt);;All Files (*)",
options=options)

        if file_path:
            # Update the passed text box with the selected file path
            text_box.setText(file_path)

    def obtain_filename(self, file_path):
        root, ext = os.path.splitext(file_path)
        return os.path.basename(root)

    def clear_layout_and_widget(self, layout):
        while layout.count():
            item = layout.takeAt(0)
            if item.layout():
                self.clear_layout_and_widget(item.layout())
            elif item.widget():
                item.widget().deleteLater()

    def resizeEvent(self, event):
        """Handles window resizing to maintain the aspect ratio of
the video."""
        self.update_image()

    def closeEvent(self, event):
        try:
            self.top_window.close()
        except:
            pass
        event.accept()

class VideoProcessor():
    def __init__(self, parent, input_source, DetectTarget,
DetectLeaf):
        super().__init__
        self.parent = parent  # Store the parent

        self.input_souce = input_source
        self.cap = cv2.VideoCapture(input_source)
        self.is_webcam = (input_source == 0)
        self.DetectTarget = DetectTarget
        self.DetectTarget.result_ready.connect(self.handle_target_res
ult)
        self.DetectLeaf = DetectLeaf
        self.DetectLeaf.result_ready.connect(self.handle_leaf_result)

        self.plant_data = []
```

```
    def run(self):
        ret, frame = self.cap.read()
        if ret:
            if self.is_webcam:
                frame = cv2.flip(frame, 1)
            self.process_frame(frame)

        else:
            print('Cannot capture frame, resetting capture.')
            self.cap.release()
            self.cap = cv2.VideoCapture(self.input_souce)
            if not self.cap.isOpened():
                self.cap.set(cv2.CAP_PROP_POS_FRAMES, 0)

    def process_frame(self, frame):
        if not self.DetectTarget.isRunning():
            self.DetectTarget.set_data(frame)
            self.DetectTarget.reset()
            self.DetectTarget.start()

        bounding_box_details = [
        {'bounding_box': data['bounding_box'],
        'label': data['label'],
        'confidence': data['confidence']}
        for data in self.plant_data]

        frame = draw_bounding_boxes(frame, bounding_box_details)
        self.parent.update_image(frame)

    def handle_target_result(self, result):
        self.plant_data, image_used_to_detect, _ = result
        if not self.DetectLeaf.isRunning():
            self.DetectLeaf.set_data(self.plant_data,
image_used_to_detect)
            self.DetectLeaf.reset()
            self.DetectLeaf.start()

    def handle_leaf_result(self, result):
        plant_detail_data, image_used_to_detect, _ = result
        self.parent.update_gallery(plant_detail_data)

class DirectoryProcessor():
    def __init__(self, parent, input_source, DetectTarget,
DetectLeaf):
        super().__init__
        self.parent = parent  # Store the parent
        self.input_source = input_source

        self.DetectTarget = DetectTarget
```

```
        self.DetectTarget.result_ready.connect(self.handle_target_res
ult)
        self.DetectLeaf = DetectLeaf
        self.DetectLeaf.result_ready.connect(self.handle_leaf_result)

        self.cache_folder = 'cache'
        self.image_paths = []

        self.current_index = 0

    def run(self):
        image_paths = [os.path.join(self.input_source, f)
                            for f in os.listdir(self.input_source)
                            if f.lower().endswith(('.png', '.jpg',
'.jpeg', '.bmp'))]
        self.image_paths = natsort.natsorted(image_paths)
        if self.image_paths:
            self.load_image()

    def load_image(self):
        image_paths = self.image_paths
        current_index = self.current_index

        # Preload previos 5 and next 5 photos
        indices_to_preload = self.get_indices_to_preload(image_paths,
current_index)
        print(f"{image_paths[current_index]}: {current_index},
{indices_to_preload}")
        self.parent.update_image_detail(f"{current_index+1} /
{len(image_paths)}")

        images_to_be_process_queue = Queue()
        if indices_to_preload:
            for indice in indices_to_preload:
                image_path = image_paths[indice]
                images_to_be_process_queue.put(image_path)

        self.process_image(image_paths[current_index],
images_to_be_process_queue)

    def process_image(self, image_to_be_show,
images_to_be_process_queue):
        if self.is_cached(image_to_be_show):
            cache_path = self.get_cache_path(image_to_be_show)
            plant_detail, image_use_to_detect, _ =
self.load_cache(cache_path)
            bounding_box_details = [
                {'bounding_box': data['plant_location'],
                'label': data['plant_label'],
```

```
                'confidence': data['plant_confidence']
                } for data in plant_detail]

            image = draw_bounding_boxes(image_use_to_detect,
bounding_box_details)
            self.parent.update_image(image)
            self.parent.update_gallery(plant_detail)

        else:
            self.parent.update_image('loading')
            self.parent.update_gallery([])

        if not images_to_be_process_queue.empty():
            image_path = images_to_be_process_queue.get()
            image = cv2.imread(image_path)
            self.DetectTarget.set_data(image, image_path)
            if not self.DetectTarget.isRunning():
                self.DetectTarget.reset()
                self.DetectTarget.start()

    def handle_target_result(self, result):
        plant_data, image_used_to_detect, image_path = result
        self.DetectLeaf.set_data(plant_data, image_used_to_detect,
image_path)
        if not self.DetectLeaf.isRunning():
            self.DetectLeaf.reset()
            self.DetectLeaf.start()

    def handle_leaf_result(self, result):
        plant_details, image_used_to_detect, image_path = result

        save_folder = self.cache_folder
        if not os.path.exists(save_folder):
            os.makedirs(save_folder)

        filename = self.obtain_filename(image_path)
        pkl_path = os.path.join(save_folder, f"{filename}.pkl")
        with open(pkl_path, 'wb') as f:
            pickle.dump(result, f)

        self.load_image()

    def get_cache_path(self, file_path):
        """Generate a cache path for the given photo."""
        cache_dir = self.cache_folder
        filename = self.obtain_filename(file_path)
        pkl_name = f"{filename}.pkl"
        return os.path.join(cache_dir, pkl_name)
```

```
    def is_cached(self, file_path):
        """Check if a photo is already cached."""
        return os.path.exists(self.get_cache_path(file_path))

    def cache_photo(self, photo_path, processed_image):
        """Save the processed image to the cache."""
        processed_image.save(self.get_cache_path(photo_path))

    def get_indices_to_preload(self, image_paths, current_index,
preload_range=2):
        """Generate indices to preload based on the current index and
preload range."""
        indices_to_preload = []

        # Calculate the range of indices to preload
        for i in range(-preload_range, preload_range + 1):
            index = self.get_index(current_index + i, image_paths)
            # Append the index if it's within the valid range of
image_paths
            if 0 <= index < len(image_paths):
                image_path = image_paths[index]
                if not self.is_cached(image_path):
                    indices_to_preload.append(index)

        if current_index in indices_to_preload:
            indices_to_preload.remove(current_index)  # Remove the
number from its current position
            indices_to_preload.insert(0, current_index)  # Insert the
number at the beginning of the list

        return indices_to_preload

    def get_index(self, index, image_paths):
        return (index) % len(image_paths)

    def load_cache(self, cache_path):
        with open(cache_path, 'rb') as f:
            data = pickle.load(f)
        return data

    def show_previous_image(self):
        if self.image_paths:
            self.current_index = self.get_index(self.current_index -
1, self.image_paths)
            self.DetectLeaf.clear_queue()
            self.load_image()

    def show_next_image(self):
        if self.image_paths:
```

```
            self.current_index = self.get_index(self.current_index + 
1, self.image_paths)
            self.DetectLeaf.clear_queue()
            self.load_image()

    def obtain_filename(self, file_path):
        root, ext = os.path.splitext(file_path)
        return os.path.basename(root)

class TopWindow(QWidget):
    def __init__(self, plant_details):
        super().__init__()

        self.setWindowTitle("Plant Details")
        self.setGeometry(100, 100, 1200, 600)
        self.setAttribute(Qt.WA_DeleteOnClose)  # Ensure the window 
is deleted when closed

        # Set the main layout for the TopWindow
        main_layout = QHBoxLayout(self)
        self.setLayout(main_layout)

        # Create plant_layout
        plant_layout, self.plant_image_label, 
self.plant_detail_container = self.create_plant_details_display()

        # Create tab
        tab_layout, tab_widget = self.create_tabs()
        scroll_area, self.content_Layout = self.create_scroll_area()
        self.large_plant_image_label = self.create_image_container()
        self.large_plant_image_label.setMinimumSize(500, 500)
        pie_chart_widget_area, pie_chart_content_Layout = 
self.create_layout_with_frame()

        # Add tabs to the QTabWidget
        tab_names = ["Details", "Image", "Chart"]
        widgets = [scroll_area, self.large_plant_image_label, 
pie_chart_widget_area]
        for widget, tab_name in zip(widgets, tab_names):
            tab_widget.addTab(widget, tab_name)

        # Connect the currentChanged signal to a custom slot
        tab_widget.currentChanged.connect(self.on_tab_changed)

        # Add layout to main_layout
        main_layout.addLayout(plant_layout)
        main_layout.addLayout(tab_layout)

        # Process_plant_details
```

```
        self.display_plant_details(plant_details)

        # Show Chart
        chart_view = self.create_chart(plant_details)
        pie_chart_content_Layout.addWidget(chart_view)

    def display_plant_details(self, plant_details):
        # Plant details
        plant_datas = {'Name': plant_details['plant_label'],
                  'Height': f"{plant_details['plant_height']:.4f} cm",
                  'Distance': f"{plant_details['plant_distance']:.4f}
cm",
                  'Leaf count': plant_details['leaf_count'],
                  'Unhealthy Leaf count':
plant_details['unhealth_leaf_count'],
                  'Health': plant_details['plant_health_status'],
                  'Disease': plant_details['plant_disease'],
                  }
        for title, data in plant_datas.items():
            details_label = QLabel(f"<b>{title}:</b> {data}")
            details_label.setWordWrap(True)  # Enable word wrapping
            details_label.setFixedWidth(500)  # Set a fixed width for
the label
            self.plant_detail_container.addWidget(details_label)

        bounding_box_details = []
        for leaf_detail in plant_details['leaf_detail']:
            # Get leaf bounding box details
            bounding_box_details.append(
                {'bounding_box': leaf_detail['bounding_box'],
                'label': leaf_detail['label'],
                'confidence': leaf_detail['confidence']}
                )

            # Create gallery container
            leaf_image_label, gallery_container,
leaf_detail_container = self.create_gallary_container()

            # Preprocess input image
            x1, y1, x2, y2 = leaf_detail['bounding_box']
            plant_img = plant_details['plant_image']
            leaf_image = plant_img[y1:y2, x1:x2]
            pixmap = preprocess_input(leaf_image, 250, 250)

            # insert preproccessed image to plant_image_label
            leaf_image_label.setPixmap(pixmap)

            # Add the row layout to the scrollable layout
            self.content_Layout.addLayout(gallery_container)
```

```
            leaf_datas = {'Area': f"{leaf_detail['area']:.4f} cm²",
                        'Health': leaf_detail['health'],
                        'Disease': leaf_detail['disease']
                        }
            for title, data in leaf_datas.items():
                leaf_details_label = QLabel(f"<b>{title}:</b>
{data}")
                leaf_detail_container.addWidget(leaf_details_label)

        plant_image =
draw_bounding_boxes(plant_details['plant_image'],
bounding_box_details)
        pixmap = preprocess_input(plant_image, 500, 500)
        self.plant_image_label.setPixmap(pixmap)
        self.update_image(plant_image)

    def update_image(self,image=None):
        if image is not None:
            self.current_image = image
        if hasattr(self,"current_image"):
            pixmap = preprocess_input(self.current_image,
self.large_plant_image_label.width()-2,
self.large_plant_image_label.height()-2)
            self.large_plant_image_label.setPixmap(pixmap)

    def create_image_container(self):
        # Display Plant Image
        plant_image_label = QLabel()
        plant_image_label.setFrameStyle(QFrame.Box | QFrame.Plain)
        plant_image_label.setAlignment(Qt.AlignCenter)
        return plant_image_label

    def create_plant_details_display(self):
        # Plant detail layout
        plant_layout = QVBoxLayout()
        plant_layout.setAlignment(Qt.AlignLeft | Qt.AlignTop)

        # Display Plant Image
        plant_image_label = self.create_image_container()
        plant_image_label.setFixedHeight(500)
        plant_image_label.setFixedWidth(500)

        # Create a plant detail container for the Pplant image
details
        plant_detail_container = QVBoxLayout()
        plant_detail_container.setAlignment(Qt.AlignLeft |
Qt.AlignTop)
```

```
        # Add widget & layout to plant_layout
        plant_layout.addWidget(plant_image_label)
        plant_layout.addLayout(plant_detail_container)

        return plant_layout, plant_image_label,
plant_detail_container

    def create_tabs(self):
        # Create QVBoxLayout for tab_widget
        tab_layout = QVBoxLayout()

        # Create the QTabWidget
        tab_widget = QTabWidget()
        tab_layout.addWidget(tab_widget)

        return tab_layout, tab_widget

    def create_scroll_area(self):
        scroll_area = QScrollArea()
        scroll_area.setWidgetResizable(True)

        content_widget = QWidget()
        content_Layout = QVBoxLayout(content_widget)
        content_Layout.setAlignment(Qt.AlignTop)
        scroll_area.setWidget(content_widget)

        return scroll_area, content_Layout

    def create_layout_with_frame(self):
        layout_area = QHBoxLayout()

        # Create a QWidget to hold the QHBoxLayout
        widget_area = QFrame()
        widget_area.setFrameStyle(QFrame.Box | QFrame.Plain)
        widget_area.setLayout(layout_area)

        return widget_area, layout_area

    def create_gallary_container(self):
        # Create a horizontal layout for the row
        gallery_container = QHBoxLayout()

        # Create a label for the image
        leaf_image_label = self.create_image_container()
        leaf_image_label.setFixedHeight(250)
        leaf_image_label.setFixedWidth(250)

        # Create a detail container for the image details
        leaf_detail_container = QVBoxLayout()
```

```
        leaf_detail_container.setAlignment(Qt.AlignLeft |
Qt.AlignTop)

        # Add widgets & layout to the gallery_container layout
        gallery_container.addWidget(leaf_image_label)
        gallery_container.addLayout(leaf_detail_container)

        return leaf_image_label, gallery_container,
leaf_detail_container

    def create_chart(self, plant_details):
        chart = SmartChart()
        chart.resize(700, 400)
        chart_view = SimpleChartView(chart)

        value_count = defaultdict(int)
        leaf_details_dict_list =  plant_details['leaf_detail']
        for leaf_detail_dict in leaf_details_dict_list:
            if 'disease' in leaf_detail_dict:
                value_count[leaf_detail_dict['disease']] += 1

        dict_count = dict(value_count)

        for (disease_type, count) in dict_count.items():
            if "nutritional" in disease_type.lower():
                color_hexcode = "#fd635c"
            elif "none" in disease_type.lower():
                color_hexcode = "#21ab72"
            else:
                color_hexcode = "#82d3e5"

            chart.add_slice(disease_type, count, color_hexcode)

        return chart_view

    def on_tab_changed(self, index):
        if index == 1:
            self.update_image()

    def resizeEvent(self, event):
        """Handles window resizing to maintain the aspect ratio of
the video."""
        self.update_image()

class SmartChart(QChart):
    def __init__(self, parent=None):
        """
        Initialization with layout and population
        """
```

```
        super(SmartChart, self).__init__(parent)
        self.offset = 140

        self.setBackgroundBrush(QColor(30, 30, 30))  # Dark grey
background
        self.setMargins(QMargins(0, 0, 0, 0))
        self.legend().hide()
        self.setAnimationOptions(QChart.SeriesAnimations)

        self.__outer = QPieSeries()
        self.__inner = QPieSeries()
        self.__outer.setHoleSize(0.35)
        self.__outer.setPieStartAngle(self.offset)
        self.__outer.setPieEndAngle(self.offset+360)
        self.__inner.setPieSize(0.35)
        self.__inner.setHoleSize(0.3)
        self.__inner.setPieStartAngle(self.offset)
        self.__inner.setPieEndAngle(self.offset+360)

        self.addSeries(self.__outer)
        self.addSeries(self.__inner)

    def clear(self):
        """
        Clear all slices in the pie chart
        """
        for slice_ in self.__outer.slices():
            self.__outer.take(slice_)

        for slice_ in self.__inner.slices():
            self.__inner.take(slice_)

    def add_slice(self, name, value, color):
        """
        Add one slice to the pie chart

        :param name: str. name of the slice
        :param value: value. value of the slice (contribute to how
much the
                      slice would span in angle)
        :param color: str. hex code for slice color
        """
        # outer
        outer_slice = QPieSlice(name, value)
        outer_slice.setColor(QColor(color))
        outer_slice.setLabelBrush(QColor(color))

        outer_slice.hovered.connect(lambda is_hovered:
self.__explode(outer_slice, is_hovered))
```

```
        outer_slice.percentageChanged.connect(lambda:
self.__update_label(outer_slice, name))

        self.__outer.append(outer_slice)

        # inner
        inner_color = self.get_secondary_color(color)
        inner_slice = QPieSlice(name, value)
        self.__inner.append(inner_slice)
        inner_slice.setColor(inner_color)
        inner_slice.setBorderColor(inner_color)

    def remove_slice(self, name):
        """
        Remove a slice from the pie chart by its name

        :param name: str. name of the slice to remove
        """
        for slice_ in self.__outer.slices():
            title = self.extract_title_from_label(slice_.label())
            if title == name:
                self.__outer.take(slice_)
                break

        for slice_ in self.__inner.slices():
            title = self.extract_title_from_label(slice_.label())
            if title == name:
                self.__inner.take(slice_)
                break

    @staticmethod
    def __update_label(slice_, title):
        """
        Update the label of a slice

        :param slice_: QPieSlice. the slice the label is applied
        :param title: str. title of the label
        """
        text_color = 'white'
        font_size = '8pt'  # Adjust the font size here
        if slice_.percentage() > 0.1:
            slice_.setLabelPosition(QPieSlice.LabelInsideHorizontal)
            text_color = 'white'

        label = "<p align='center' style='color:{}; font-
size:{}'>{}<br>{}%</p>".format(
            text_color,
            font_size,
            title,
```

```
            round(slice_.percentage() * 100, 2)
        )

        slice_.setLabel(label)

        if slice_.percentage() > 0.03:
            slice_.setLabelVisible()

    @staticmethod
    def extract_title_from_label(html_label):
        """
        Extracts the title from an HTML-formatted label string.

        :param html_label: str. The HTML-formatted label string
        :return: str. The extracted title
        """
        # Define a regular expression pattern to extract text between
<p> and <br>
        pattern = re.compile(r'<p[^>]*>(.*?)<br>', re.DOTALL)
        match = pattern.search(html_label)

        if match:
            return match.group(1).strip()
        return ""

    def __explode(self, slice_, is_hovered):
        """
        Explode function slot for hovering effect

        :param slice_: QtChart.QPieSlice. the slice hovered
        :param is_hovered: bool. hover enter (True) or leave (False)
        """
        if is_hovered:
            start = slice_.startAngle()
            end = slice_.startAngle() + slice_.angleSpan()
            self.__inner.setPieStartAngle(end)
            self.__inner.setPieEndAngle(start+360)
        else:
            self.__inner.setPieStartAngle(self.offset)
            self.__inner.setPieEndAngle(self.offset+360)

        slice_.setLabelVisible(is_hovered)
        slice_.setExplodeDistanceFactor(0.1)
        slice_.setExploded(is_hovered)

        if slice_.percentage() > 0.03:
            slice_.setLabelVisible()

    @staticmethod
```

```
    def hex_to_rgb(hexcode):
        """Convert hex color code to RGB tuple."""
        from PIL import ImageColor
        return ImageColor.getcolor(hexcode, "RGB")

    @staticmethod
    def rgb_to_hex(rgb):
        """Convert RGB tuple to hex color code."""
        return '#{:02x}{:02x}{:02x}'.format(rgb[0], rgb[1], rgb[2])

    def get_secondary_color(self, hexcode_color1,
hexcode_color2="#FFFFFF", alpha=0.5):
        """
        Get secondary color which is blended 50% with white
        to appear lighter

        :param hexcode: str. color hex code starting with '#'
                        eg. ('#666666')
        :return: QtGui.QColor
        """
        # Convert hex to RGB
        rgb1 = self.hex_to_rgb(hexcode_color1)
        rgb2 = self.hex_to_rgb(hexcode_color2)

        blended_rgb = tuple(int(a * (1 - alpha) + b * alpha) for a, b
in zip(rgb1, rgb2))

        blended_hex = self.rgb_to_hex(blended_rgb)

        return QColor(blended_hex)

class SimpleChartView(QChartView):
    """
    A simple wrapper chart view, to be expanded
    """
    def __init__(self, chart):
        super(SimpleChartView, self).__init__(chart)

        self.setRenderHint(QPainter.Antialiasing)

class DetectTarget(QThread):
    result_ready = pyqtSignal(object)
    def __init__(self, TargetDetection, verbose=False):
        super().__init__()
        self.TargetDetection = TargetDetection
        self.verbose = verbose
        self.image_data = []
        self.label = ''
        self.mutex = QMutex()
```

```
    def set_data(self, image_data, label=None):
        with QMutexLocker(self.mutex):
            self.image_data = image_data
            self.label = label

    def set_verbose(self, verbose):
        self.verbose = verbose

    def reset(self):
        # Implement any necessary reset logic here
        pass

    def run(self):
        image_data = self.image_data
        label = self.label
        plant_data = []
        if np.any(image_data):
            plant_data =
self.TargetDetection.detect_plants(image_data, verbose=self.verbose)
        self.result_ready.emit((plant_data, image_data, label))  #
Emit the result

class DetectLeaf(QThread):
    result_ready = pyqtSignal(object)
    def __init__(self, TargetDetection, verbose=False,
distance_scale=10, sensor_height=24, sensor_width=35,
focal_length=30, rest=False):
        super().__init__()
        self.TargetDetection = TargetDetection
        self.verbose = verbose
        self.distance_scale = distance_scale
        self.sensor_width = sensor_width
        self.sensor_height = sensor_height
        self.focal_length = focal_length
        self.rest = rest
        self.plant_data = Queue()
        self.original_image = Queue()
        self.label = Queue()
        self.mutex = QMutex()

    def set_rest(self, rest):
        self.rest = rest

    def set_data(self, plant_data, original_image, label=None):
        with QMutexLocker(self.mutex):
            self.plant_data.put(plant_data)
            self.original_image.put(original_image)
            self.label.put(label)
```

```
    def set_verbose(self, verbose):
        self.verbose = verbose

    def set_sensor_size(self, distance_scale, sensor_height,
sensor_width, focal_length):
        self.distance_scale = distance_scale
        self.sensor_height = sensor_height
        self.sensor_width = sensor_width
        self.focal_length = focal_length

    def clear_queue(self):
        self.plant_data = Queue()
        self.original_image = Queue()
        self.label = Queue()

    def reset(self):
        # Implement any necessary reset logic here
        pass

    def run(self):
        while not self.plant_data.empty():
            plant_data = self.plant_data.get()
            original_image = self.original_image.get()
            label = self.label.get()
            plant_datail = []
            if np.any(plant_data):
                # Obtain depth map of the original image
                depth_map =
self.TargetDetection.detect_depth(original_image,
verbose=self.verbose)

                # Obtain focal length of camera in pixel
                image_height, image_width, _ = original_image.shape

                for plant_info in plant_data:
                    # Distance
                    distance_cm =
calculate_distance_of_target(depth_map, plant_info['plant_mask'],
self.distance_scale)
                    distance_cm = abs(distance_cm)

                    # obtain height and width in pixel/cm based on
distance in cm
                    height_pixel_cm, width_pixel_cm =
calculate_size_of_pixel_in_cm(distance_cm, image_height, image_width,
self.sensor_height, self.sensor_width, self.focal_length)
```

```
                    # Calculate the height of plant based on the
provided depth map and plant mask
                    height_cm =
calculate_height_cm(plant_info['plant_mask'], height_pixel_cm)

                    # Plant image
                    x1, y1, x2, y2 = plant_info['bounding_box']
                    plant_image = original_image[y1:y2, x1:x2]

                    # Detect the leafs of the given plant image
                    leaf_details = self.TargetDetection.detect_leafs(
                        plant_image,
                        height_pixel_cm, width_pixel_cm,
                        verbose=self.verbose
                        )

                    disease_list = []
                    for leaf_detail in leaf_details:
                        if 'disease' in leaf_detail:
                            if leaf_detail['disease'] != 'None':
                                disease_list.append(leaf_detail['dise
ase'])

                    plant_health_status = 'Healthy'
                    plant_disease = 'None'
                    if disease_list:
                        if (len(disease_list) / len(leaf_details)) >=
0.5:
                            plant_health_status = 'Unhealthy'

                            # Count the occurrences of each item
                            disease_counts = Counter(disease_list)

                            # Calculate the percentage for each item
                            item_percentages = {item: (count /
len(leaf_details)) * 100 for item, count in disease_counts.items()}

                            # Find the item with the highest
percentage
                            plant_disease = max(item_percentages,
key=item_percentages.get)

                    plant_datail.append({
                        'plant_id': plant_info['id'],
                        'plant_label': plant_info['label'],
                        'plant_image': plant_image,
                        'plant_location': plant_info['bounding_box'],
                        'plant_confidence': plant_info['confidence'],
                        'plant_distance': distance_cm,
```

```
                        'plant_height': height_cm,
                        'unhealth_leaf_count': len(disease_list),
                        'leaf_count': len(leaf_details),
                        'plant_health_status': plant_health_status,
                        'plant_disease': plant_disease,
                        'leaf_detail': leaf_details,
                    })
            self.result_ready.emit((plant_datail, original_image, 
label))  # Emit the result

            if self.rest:
                time.sleep(2)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = RealTimeVideoApp()
    window.show()
    sys.exit(app.exec_())
```

APPENDIX C: Code for Target Detection in Python Language

```
import time
import PIL
import cv2
import numpy as np
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QPixmap, QImage

import torch
from ultralytics import YOLO
from transformers import DPTImageProcessor, DPTForDepthEstimation
# import keras
# from sklearn.preprocessing import normalize

import multiprocessing
multiprocessing.set_start_method('spawn') # Ensure the 'spawn' method 
is used
multiprocessing.freeze_support()

class TargetDetect():
```

```
    def __init__(self, model_path_plant=None, model_path_leaf=None):

        # Check if GPU is available
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        print(f"Using device: {self.device}")

        # Load the model
        if not model_path_plant or not model_path_leaf:
            raise ValueError("Cannot load model from path")
        self.model_plant = YOLO(model_path_plant)
        self.model_plant.to(self.device)
        self.model_leaf = YOLO(model_path_leaf)
        self.model_leaf.to(self.device)

        # Load the DPT model and processor
        self.processor =
DPTImageProcessor.from_pretrained("Intel/dpt-large")
        self.model =
DPTForDepthEstimation.from_pretrained("Intel/dpt-
large").to(self.device)

    def detect_depth(self, image, verbose=True):
        start_time = time.time()
        # Convert BGR image to RGB since DPT model expects an RGB
image
        image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        # Convert the numpy array (image_rgb) to PIL Image for
processing
        image_pil = PIL.Image.fromarray(image_rgb)

        # Prepare image for the DPT model
        inputs = self.processor(images=image_pil,
return_tensors="pt").to(self.device)

        # Perform inference to get depth estimation
        with torch.no_grad():
            outputs = self.model(**inputs)
            predicted_depth = outputs.predicted_depth

        # Get the original size of the image
        original_size = image.shape[:2]  # (height, width)

        # Interpolate to the original image size
        prediction = torch.nn.functional.interpolate(
            predicted_depth.unsqueeze(1),
            size=original_size,
            mode="bicubic",
            align_corners=False,
```

```
        )

        # Convert to numpy array
        depth_map = prediction.squeeze().cpu().numpy()

        # Normalize and convert depth map to 8-bit image for display
        # depth_map_normalized = (depth_map * 255 /
np.max(depth_map)).astype("uint8")
        # depth_image = PIL.Image.fromarray(depth_map_normalized)

        end_time = time.time()
        if verbose:
            print(f"Speed depth detection: {((end_time-start_time) *
1000):.4f} ms")

        return depth_map

    def non_max_suppression_fast(self, boxes, overlapThresh):
        if len(boxes) == 0:
            return []

        if boxes.dtype.kind == "i":
            boxes = boxes.astype("float")

        pick = []

        x1 = boxes[:, 0]
        y1 = boxes[:, 1]
        x2 = boxes[:, 2]
        y2 = boxes[:, 3]

        area = (x2 - x1 + 1) * (y2 - y1 + 1)
        idxs = np.argsort(y2)

        while len(idxs) > 0:
            last = len(idxs) - 1
            i = idxs[last]
            pick.append(i)

            xx1 = np.maximum(x1[i], x1[idxs[:last]])
            yy1 = np.maximum(y1[i], y1[idxs[:last]])
            xx2 = np.minimum(x2[i], x2[idxs[:last]])
            yy2 = np.minimum(y2[i], y2[idxs[:last]])

            w = np.maximum(0, xx2 - xx1 + 1)
            h = np.maximum(0, yy2 - yy1 + 1)

            overlap = (w * h) / area[idxs[:last]]
```

```
            idxs = np.delete(idxs, np.concatenate(([last],
np.where(overlap > overlapThresh)[0])))

        return pick

    def detect_plants(self, frame, confidence=0.7, overlapThresh=0.3,
verbose=True):
        results = self.model_plant.predict(source=frame,
conf=confidence, save=False, stream=False, retina_masks=True,
device=self.device, verbose=verbose, cache=False)

        boxes = []
        class_indices = []
        contours = []
        confidences = []

        for result in results:
            for ci, c in enumerate(result):
                box =
c.boxes.xyxy.cpu().numpy().squeeze().astype(np.int32)
                cls_idx = int(c.boxes.cls.tolist().pop())
                confidence = c.boxes.conf.tolist().pop()
                contour = c.masks.xy[0].astype(np.int32).reshape(-1,
1, 2)
                # contour = result.masks.xy[0]
                boxes.append(box)
                class_indices.append(cls_idx)
                confidences.append(confidence)
                contours.append(contour)

        # Convert boxes to numpy array
        boxes = np.array(boxes)

        # Perform NMS
        plant_data = []

        if len(boxes) > 0:
            indices = self.non_max_suppression_fast(boxes,
overlapThresh)
            if verbose:
                print(f'Boxes shape: {boxes.shape}, NMS indices:
{indices}')  # Debugging info

            for idx in indices:
                x1, y1, x2, y2 = boxes[idx]
                label = self.model_plant.names[class_indices[idx]]
                confidence = confidences[idx]
                bounding_box = boxes[idx]
                contour = contours[idx]
```

```
                # # Create contour mask
                b_mask = np.zeros(frame.shape[:2], np.uint8)

                # Fill the mask in the binary mask
                binary_mask = cv2.fillPoly(b_mask, [contour], 255)

                # Add data to plant_data list
                plant_data.append({
                    'id': idx,
                    'label': label,
                    'confidence': confidence,
                    'bounding_box': bounding_box,
                    'plant_mask': binary_mask,
                    })

        return plant_data

    def detect_leafs(self, plant_image, height_pixel_cm,
width_pixel_cm, confidence=0.5, overlapThresh=0.3, verbose=True):
        results = self.model_leaf.predict(source=plant_image,
conf=confidence, save=False, stream=True, retina_masks=True,
device=self.device, verbose=verbose, cache=False)

        boxes = []
        class_indices = []
        contours = []
        confidences = []

        for result in results:
            for ci, c in enumerate(result):
                box =
c.boxes.xyxy.cpu().numpy().squeeze().astype(np.int32)
                cls_idx = int(c.boxes.cls.tolist().pop())
                confidence = c.boxes.conf.tolist().pop()
                boxes.append(box)
                class_indices.append(cls_idx)
                confidences.append(confidence)
                contours.append(c.masks.xy[0].astype(np.int32).reshap
e(-1, 1, 2))

        # Convert boxes to numpy array
        boxes = np.array(boxes)

        # Perform NMS
        leaf_data = []

        if len(boxes) > 0:
```

```
            indices = self.non_max_suppression_fast(boxes,
overlapThresh)
            if verbose:
                print(f'Boxes shape: {boxes.shape}, NMS indices:
{indices}')  # Debugging info

            for idx in indices:
                x1, y1, x2, y2 = boxes[idx]
                label = self.model_leaf.names[class_indices[idx]]
                confidence = confidences[idx]
                bounding_box = boxes[idx]
                contour = contours[idx]

                # Leaf iamge
                leaf_image = plant_image[y1:y2, x1:x2]

                # Initialize the mask with zeros
                mask = np.zeros(plant_image.shape[:2],
dtype=np.uint8)

                # Draw contour on the mask
                binary_mask = cv2.fillPoly(mask, [contour],
color=255)

                # Calculate leaf area in cm2
                leaf_area = calculate_area_cm2(binary_mask,
height_pixel_cm, width_pixel_cm)

                if label == 'Ginger-Leaf_Healthy':
                    # Add data to plant_data list
                    leaf_data.append({
                        'id': idx,
                        'label': label.replace("Ginger-Leaf_", ""),
                        'confidence': confidence,
                        'bounding_box': bounding_box,
                        'area': leaf_area,
                        'health': 'Healthy',
                        'disease': 'None'
                        })

                else:
                    # Add data to plant_data list
                    leaf_data.append({
                        'id': idx,
                        'label': label.replace("Ginger-Leaf_", ""),
                        'confidence': confidence,
                        'bounding_box': bounding_box,
                        'leaf_image': leaf_image,
                        'area': leaf_area,
```

```
                        'health': 'Unhealthy',
                        'disease': label.replace("Ginger-Leaf_", "")
                        })

        return leaf_data

def preprocess_input(image_data, target_width=None,
target_height=None):
    # Convert BGR to RGB for compatibility with Qt
    image = cv2.cvtColor(image_data, cv2.COLOR_BGR2RGB)

    # Extract image dimensions
    height, width, channel = image.shape

    # Calculate bytes per line for QImage creation
    bytes_per_line = 3 * width

    # Create QImage from the image data
    q_img = QImage(image.data, width, height, bytes_per_line,
QImage.Format_RGB888)

    # Convert QImage to QPixmap for display on the label
    pixmap = QPixmap.fromImage(q_img)

    if target_width and target_height:
        # Resize the pixmap to fit the label's dimensions while
maintaining aspect ratio
        pixmap = pixmap.scaled(target_width, target_height,
Qt.KeepAspectRatio)

    return pixmap

def draw_bounding_boxes(image_data, bounding_box_details):
    # Create a copy to avoid modifying the original image
    image_data_copy = image_data.copy()

    # Calculate the bounding box thickness based on the image size
    height, width = image_data_copy.shape[:2]
    thickness = max(1, int(min(height, width) / 200))  # Adjust the
divisor for different thicknesses

    # Calculate text size and adjust the font scale based on the
image size
    font_scale = min(height, width) / 600  # Adjust the divisor for
different font sizes

    for bounding_box_detail in bounding_box_details:
        x1, y1, x2, y2 = bounding_box_detail['bounding_box']
```

```
        cv2.rectangle(image_data_copy, (x1, y1), (x2, y2), (0, 255,
0), thickness)
        cv2.putText(image_data_copy,
f"{bounding_box_detail['label']}:
{bounding_box_detail['confidence']:.2f}", (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, font_scale , (0, 255, 0), thickness)

    return image_data_copy

def calculate_distance_of_target(depth_map, mask, scale_factor=10):
    # Ensure depth_map is float
    depth_map = depth_map.astype(float)

    # Ensure mask is binary (convert to boolean array if needed)
    mask = mask > 0

    # Apply the mask to the depth map
    masked_depth = np.where(mask, depth_map, np.inf)  # Set
background pixels (outside the mask) to infinity

    # Find the minimum value inside the masked region (the closest
distance)
    closest_distance = np.min(masked_depth)

    if closest_distance == np.inf:
        return 0

    return closest_distance * scale_factor

def calculate_size_of_pixel_in_cm(distance_cm, image_height_pixels,
image_width_pixels, sensor_height_mm=24, sensor_width_mm=36,
focal_length_mm=15):
    fov_height_rad = 2 *
np.arctan(sensor_height_mm/(2*focal_length_mm))
    height_pixel_cm = np.tan( (fov_height_rad) /2) *
(distance_cm/image_height_pixels)

    fov_width_cm = 2 * np.arctan(sensor_width_mm/(2*focal_length_mm))
    width_pixel_cm = np.tan( (fov_width_cm) /2) *
(distance_cm/image_width_pixels)

    return height_pixel_cm, width_pixel_cm

def calculate_height_cm(mask, height_pixel_cm):
    # Get the topmost and bottommost points of the mask
    y_indices, x_indices = np.where(mask > 0)
    if len(y_indices) == 0:  # Ensure there are mask pixels detected
        return 0
    top_y = np.min(y_indices)
```

```
    bottom_y = np.max(y_indices)

    # Calculate the pixel height
    height_pixel = bottom_y - top_y

    # Convert pixel height to real-world height
    height_cm = height_pixel * height_pixel_cm

    # print(f"height_pixel = {height_pixel}, height_cm = {height_cm},
")

    return height_cm

def calculate_area_cm2(mask, pixel_height_cm, pixel_width_cm):
    # Compute the area of the segmented object in pixels
    area_pixels = np.sum(mask == 255)

    # Calculate real-world dimensions
    area_cm2 = (pixel_height_cm * pixel_width_cm) * area_pixels

    return area_cm2
```

APPENDIX D: Test Image used in Evaluated Depth Estimation Model

| **Test Image** | **Depth Map** | $y_i$ **(cm)** | $\hat{y}_i$ **(cm)** | $\lvert y_i - \hat{y}_i \rvert$ | $(y_i - \hat{y}_i)^2$ |
|---|---|---|---|---|---|
| | | 50 | 48.60 | 1.40 | 1.97 |

| | | | | | |
|---|---|---|---|---|---|
| | | 60 | 50.83 | 9.17 | 84.10 |
| | | 70 | 72.23 | 2.23 | 4.96 |
| | | 80 | 88.45 | 8.45 | 71.41 |
| | | 90 | 78.20 | 11.80 | 139.27 |
| | | 100 | 104.38 | 4.38 | 19.17 |

| | | | | | |
|---|---|---|---|---|---|
| | | 110 | 90.12 | 19.88 | 395.36 |
| | | 120 | 113.05 | 6.95 | 48.29 |
| | | 130 | 98.52 | 31.48 | 991.08 |
| | | 140 | 114.64 | 25.36 | 643.36 |
| | | 150 | 135.23 | 14.77 | 218.25 |

| | | | | | |
|---|---|---|---|---|---|
| | | 160 | 137.04 | 22.96 | 526.94 |
| | | 170 | 131.89 | 38.11 | 1452.26 |
| | | 180 | 137.43 | 42.57 | 1812.10 |
| | | 190 | 142.44 | 47.56 | 2261.95 |