

Plantdis

Integrating Mobile Apps and Machine Learning for Plant Disease Diagnosis



Jinniu Du u7667862@anu.edu.au
Australian National University & APPN
08/2024

Abstract

Supported by the Australian Plant Phenomics Network (APPN), this project integrates advanced machine learning models into a mobile app for rapid and accurate plant disease diagnosis. Using transfer learning with MobileNetV2 and training on two open-source datasets, the models are converted to TensorFlow Lite and embedded into a Flutter-based app. This approach combines AI technology with mobile accessibility, offering an innovative, real-time solution for plant health monitoring, aiming to boost agricultural productivity and botanical research.

Contents

1. Introduction.....	6
1.1 Project Background	6
1.2 Project Purpose.....	7
2. Technologies overview.....	8
2.1 Transfer Learning.....	8
2.1.1 Selection of pre-trained model.....	8
2.1.2 Process of transfer learning.....	12
2.2 Flutter.....	14
2.3 Firebase	15
3. Architecture of software.....	17
3.1 Transfer Learning.....	18
3.2 flutter Application	20
3.3 Firebase	21
4. Opensource Dataset.....	22
4.1 Cassava_dataset	22
4.2 Plant_leave_diseases dataset	25
5. Transfer learning	28
5.1 pre-Process	28
5.1.1 Cassava_dataset pre-process.....	28

5.1.2 Plant_leave_diseases dataset pre-process	31
5.2 Training Process	34
5.2.1 Plantdis model	35
5.2.2 Cropnet model	40
6. Application.....	44
6.1 Login and Register Page.....	45
6.2 Main page	48
6.3 Result Page.....	50
6.4 Settings Page.....	54
6.4.1. Dark Mode.....	55
6.4.2. Text-to-Speech (TTS) Mode.....	55
6.5 Database	56
7. Testing	59
7.1 Testing with python	59
7.1.1 PlantDis Model result.....	59
7.1.2 CropNet Model result	64
7.2 Testing with App	67
8. Future Work.....	70
8.1 Integration of Segmentation Models.....	70
8.2 Improving Model Generalization	71

8.2.1 Dataset Expansion:.....	71
8.2.2 Model Enhancement:.....	72
9. Conclusion	73
10. Reference:.....	74

1. Introduction

1.1 Project Background

The PlantDis application is a mobile solution designed to swiftly diagnose plant diseases by analysing photos of plant leaves. Leveraging advanced machine learning models, PlantDis provides farmers and gardeners with an efficient tool for identifying plant health issues and taking prompt action.

Strategically located within the Plant Science Division at ANU, the project benefits from the diverse expertise available within the ANU community, including collaborations with the ARC Centre of Excellence in Plant Energy Biology (PEB), the ARC Centre of Excellence for Translational Photosynthesis (TP), the ANU-CSIRO Centre for Biodiversity Analysis, and the Centre of Excellence for Robotic Vision. These collaborations, combined with access to the National Computational Infrastructure (NCI) for high-performance computing, cloud services, and data repository, provide the technological foundation for the PlantDis application.

1.2 Project Purpose

The primary objective of the PlantDis project is to develop an innovative mobile application that enables rapid and convenient diagnosis of plant leaf health. This project is undertaken with the support of the Australian Plant Phenomics Network (APPN) and builds upon the extensive expertise of the Plant Phenomics Group at the Australian National University (ANU). By leveraging advanced machine learning techniques, the PlantDis application aims to provide accurate and immediate diagnostic results from captured leaf images.

This project is positioned at the forefront of innovation in plant phenomics, bioinformatics, and data visualization, areas where the Plant Phenomics Group has a strong track record. The application will serve as a critical tool for farmers, agricultural laboratories, botanical research facilities, and educational institutions, assisting them in maintaining plant health and enhancing agricultural productivity. The project aligns with APPN's mission to support groundbreaking plant research and the development of open-source, high-throughput phenomics infrastructure. The PlantDis app will contribute to the creation of open data sets for plant science researchers both nationally and internationally.

2. Technologies overview

2.1 Transfer Learning

2.1.1 Selection of pre-trained model

In our pursuit of deploying an efficient and accurate model for mobile applications, we considered four potential models: EfficientNetB2, MobileNetV2, MobileNetV3Small, and MobileNetV3Large. Recognizing the constraints of mobile devices, such as processing power and storage, we aimed to identify a model that offers a balance between high accuracy and fast inference times.

To make an informed decision, we subjected the PlantDis dataset to a uniform preprocessing step. This involved resizing the images to a consistent size and ensuring that the data was prepared for optimal input into the models. As the Fig 1 shows, we performed a controlled experiment by freezing the layers of each pre-trained model and conducting a brief training session over three epochs.

Model	Accuracy(%)	Time (seconds)	file size	Mobile_App Adaptability
EfficientNetB2	99.24	9.2400s	Big	Median
MobileNetV2	98.00	0.0781s	Small	Good
MobileNetV3Small	76.00	0.5200s	Small	Good
MobileNetV3Large	50.00	0.5351	Small	Good

Figure 1

The results, as summarized in the table, highlight several key factors:

- **Accuracy:** While EfficientNetB2 achieved the highest accuracy at 99.24%, it was significantly slower and had a larger file size, making it less ideal for mobile deployment. On the other hand, MobileNetV2 achieved a high accuracy of 98.00% with a very fast inference time, making it a strong candidate.
- **Inference Time:** MobileNetV2 outperformed all other models in terms of speed, taking only 0.0781 seconds per inference. This quick response time is crucial for real-time applications.
- **File Size and Adaptability:** While both MobileNetV3Small and MobileNetV3Large showed good adaptability for mobile use due to their smaller file sizes, their lower accuracy (76.00% and 50.00% respectively) compared to MobileNetV2 made them less favourable.

As the graph illustrates the accuracy progression of four models—EfficientNetB2, MobileNetV2, MobileNetV3Small, and MobileNetV3Large—over three epochs of training. As observed, both EfficientNetB2 and MobileNetV2 demonstrate high and consistent accuracy levels right from the first epoch, with EfficientNetB2 slightly outperforming MobileNetV2. However, considering the other factors such as inference time and mobile adaptability, MobileNetV2 emerges as the most balanced choice.

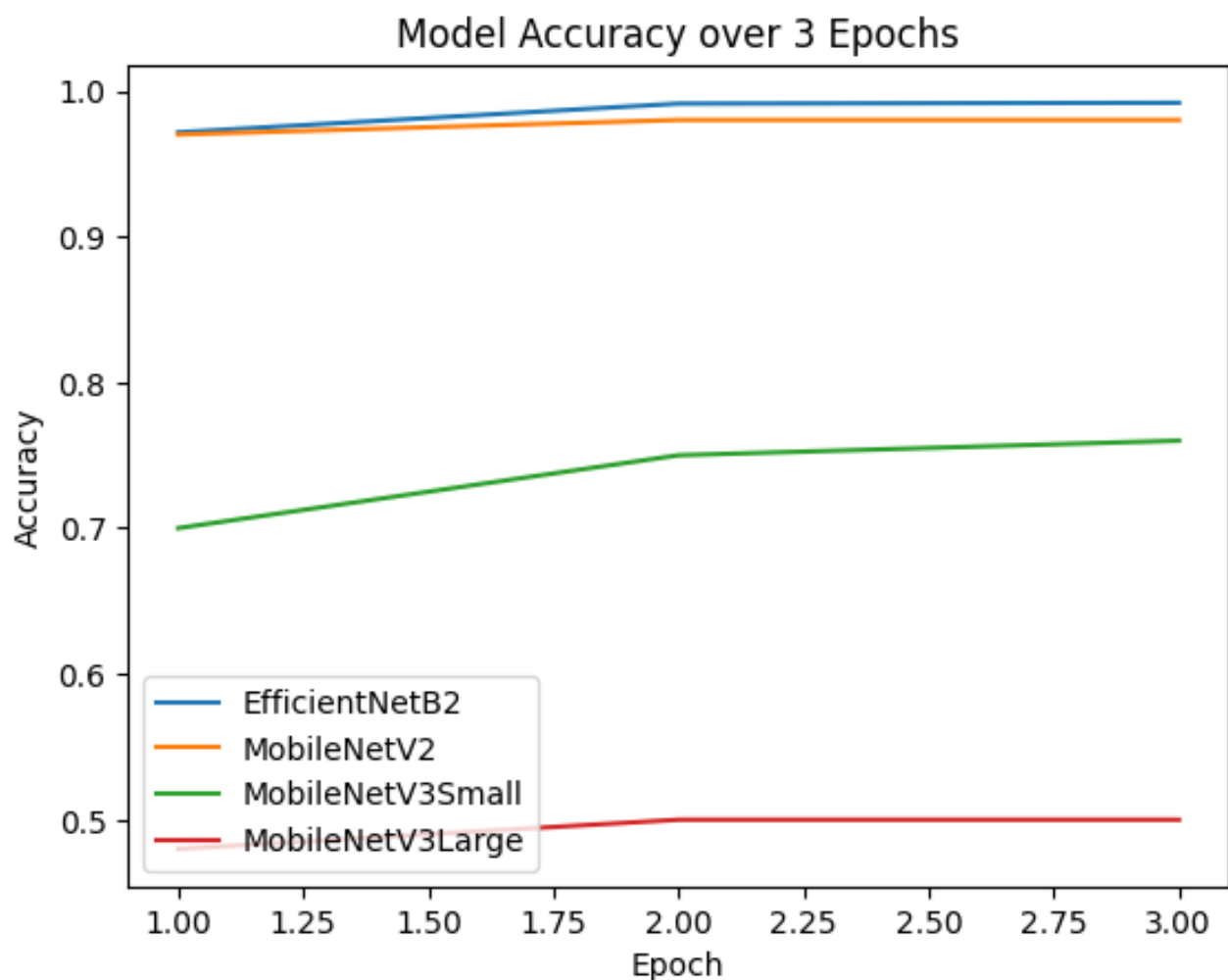


Figure 2

MobileNetV3Small shows moderate improvement over the epochs but does not reach the accuracy levels of EfficientNetB2 or MobileNetV2. Meanwhile, MobileNetV3Large starts at a significantly lower accuracy and shows minimal improvement, making it less suitable for our needs.

Given these results, we ultimately selected MobileNetV2 as the most appropriate model for our PlantDis mobile application. It offers a compelling combination of high accuracy, efficient performance, and adaptability for mobile deployment, making it well-suited for real-time plant disease detection in a mobile environment. The efficiency of MobileNetV2, both in terms of computational requirements and storage, further solidifies its selection as the optimal model for our application.

2.1.2 Process of transfer learning

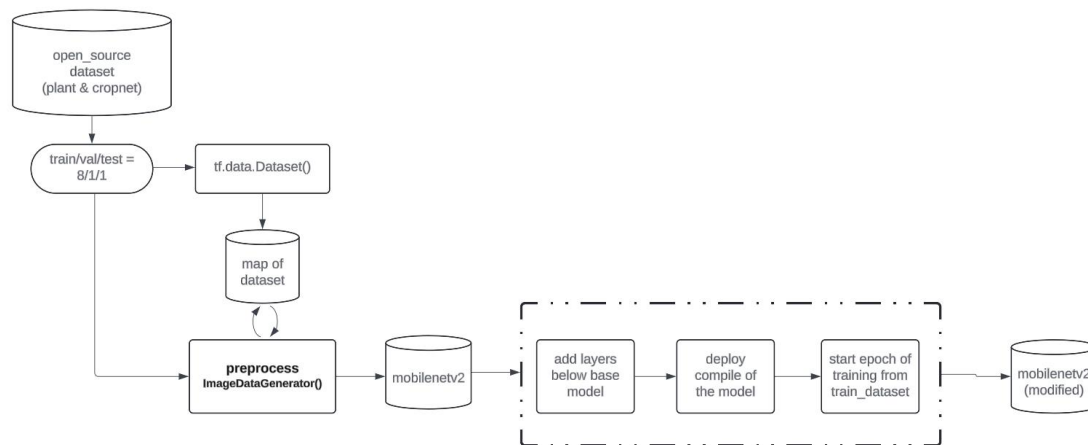


Figure 3

We did the process of integrating custom layers into a pre-trained MobileNetV2 model as part of our transfer learning approach. Fig 3 show the process includes the preparation of the dataset, the initialization and modification of the MobileNetV2 model, and the training of the model with added layers.

- Data Preparation

We utilized an open-source dataset comprising plant and crop images. The dataset was divided into training, validation, and test sets in an 8:2 ratio.

- Preprocessing

Preprocessing was carried out using the ImageDataGenerator class to perform rescaling, augmentation, and normalization. This ensures that the images fed into the model are standardized and augmented to improve the model's robustness.

- Loading Pre-trained MobileNetV2

The MobileNetV2¹ model was loaded with pre-trained weights from ImageNet. The top layers were excluded to allow the addition of custom layers specific to our classification task. As shown in Figure 5 from the original MobileNetV2 paper, MobileNetV2 demonstrates a significant improvement in performance over its predecessor, MobileNetV1, as well as other architectures like ShuffleNet and NasNet across different input resolutions. The model's efficient architecture makes it a suitable choice for mobile and embedded applications, such as the PlantDis app.

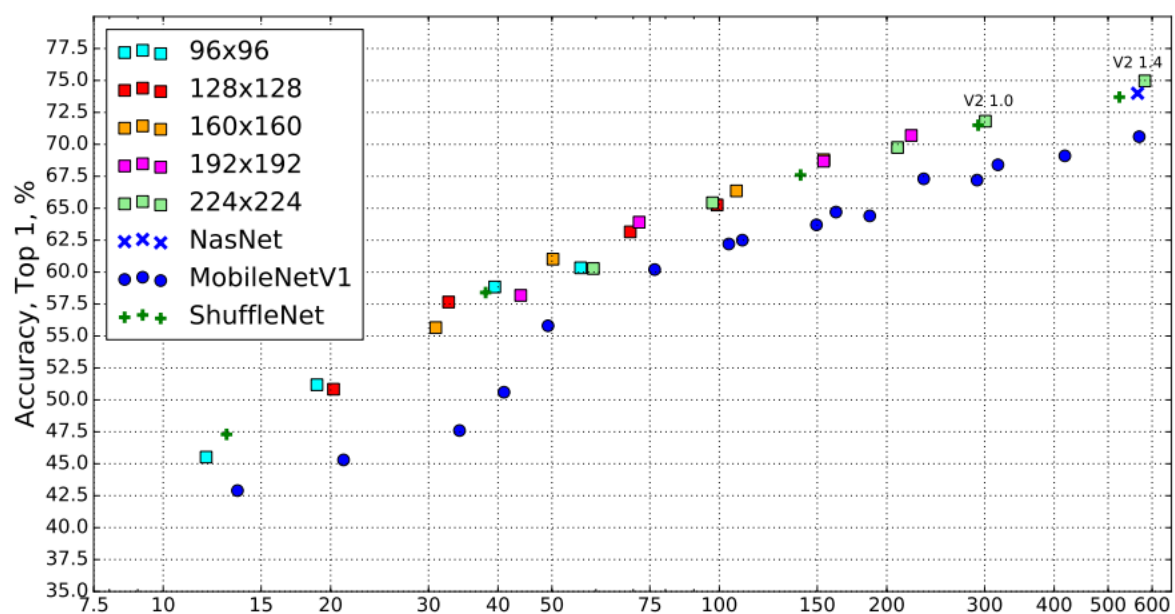


Figure 4

¹ Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, 4510–4520. <https://arxiv.org/abs/1801.04381>.

2.2 Flutter

Flutter² is an open-source UI software development toolkit created by Google. It enables the creation of natively compiled applications for mobile, web, and desktop from a single codebase. In the PlantDis application, Flutter is utilized to construct a highly responsive and visually appealing user interface. This interface allows users to seamlessly capture and upload images of plant leaves, view diagnostic results, and navigate through various features with ease.

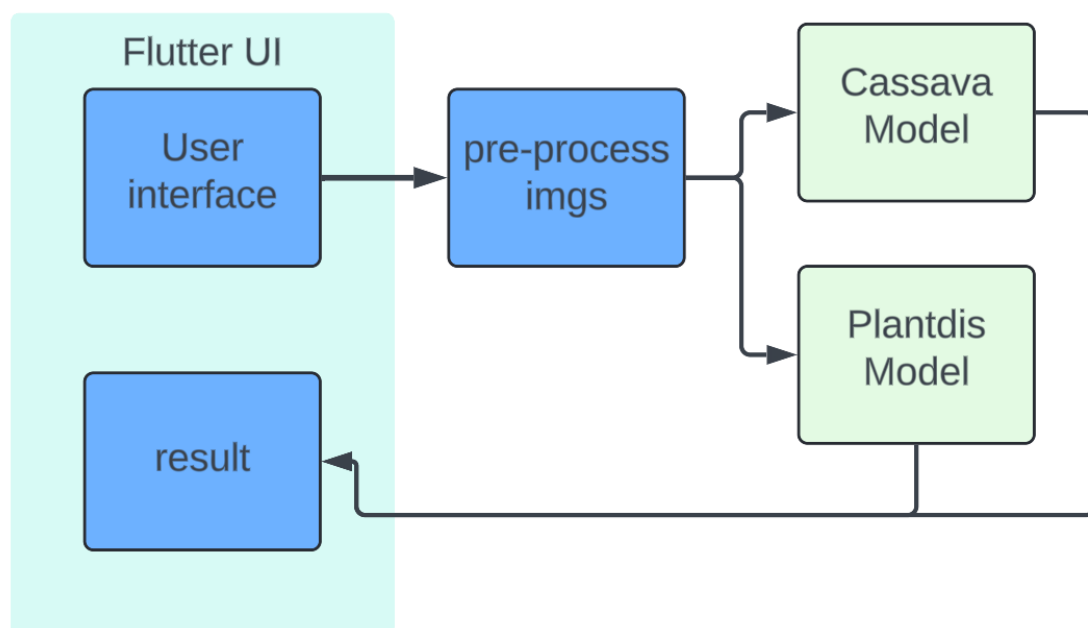


Figure 5

² Flutter: Cross-template toolkits: https://flutter.dev/multi-platform?gad_source=1&gclid=CjwKCAjw2dG1BhB4EiwA998cqE4ZlAbX595xLOfzUg6kF1gf6T38V7lrNdySvLOZvdnJd9PzMvVYhoChUQQAvD BwE&gclsrc=aw.ds

The application's front end is built using Flutter, which communicates with the MobileNetV2 model embedded in the app for image processing and classification tasks. This integration allows for real-time analysis on the device, ensuring quick and efficient processing. Flutter's extensive widget library, along with its capability to deliver high-performance applications across multiple platforms, makes it an ideal choice for the PlantDis application.

2.3 Firebase

Firebase³, developed by Google, is a comprehensive platform designed to streamline the development of high-quality apps by providing an array of tools and services. These tools include real-time databases, user authentication, cloud storage, analytics, hosting, and more. Firebase simplifies backend management, allowing developers to focus more on front-end development and user experience. The platform supports seamless integration with mobile and web applications, offering real-time updates and secure data management, which are essential for scaling applications and ensuring robust performance.

³ Firebase: Realtime database: <https://firebase.google.com/docs>

3. Data Management with Firebase Firestore

Firebase Firestore is used for storing user-specific data, such as previous diagnostic results, user profiles, and any feedback they provide about the app. Firestore is a NoSQL cloud database that is highly scalable, allowing the PlantDis application to store and sync data across all users in real-time. This ensures that user data is always up-to-date and accessible whenever needed.

4. Result Storage and Feedback

After processing the image and generating diagnostic results, the results are stored back in Firebase Firestore. As the Fig6 shows, this allows users to review their previous diagnostics, and it enables the app to provide personalized insights over time. Users can also provide feedback on the accuracy of the diagnosis, which is also stored in Firestore.

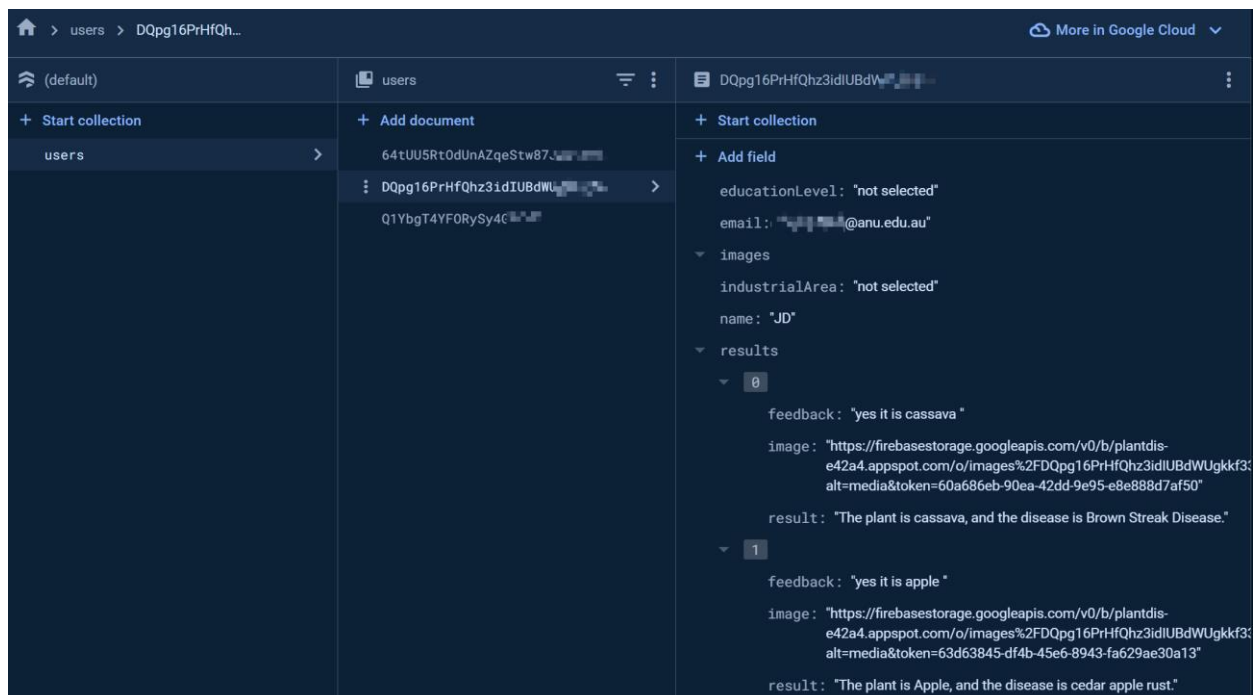


Figure 6

3. Architecture of software

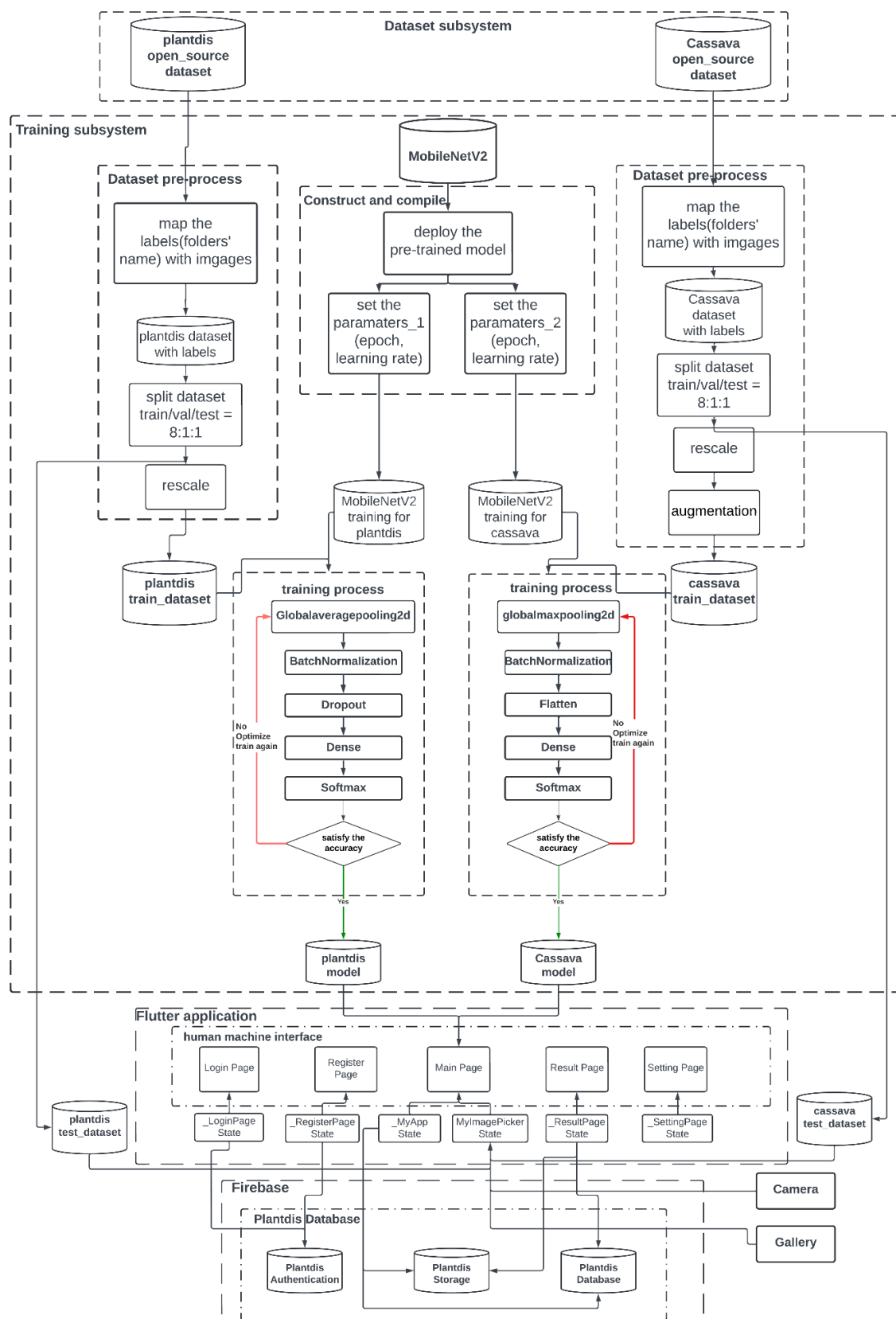


Figure 7

Fig 7 shows our system architecture and it is divided into three main subsystems: **Dataset**, **Training**, and **Flutter Application**, all integrated with **Firebase** for backend support.

1. **Dataset Subsystem:** This subsystem manages the collection and preprocessing of datasets sourced from open-source repositories. These datasets are then prepared to be passed onto the Training subsystem.
2. **Training Subsystem:** Utilizing Transfer Learning with the MobileNetV2 model, this subsystem fine-tunes the model on the preprocessed datasets. The output of this process is two trained models, one for the PlantDis dataset and another for the Cassava dataset, both optimized for mobile deployment.
3. **Flutter Application Subsystem:** The trained models are integrated into a mobile application developed using Flutter. The application provides a user-friendly interface, allowing users to upload plant images for disease diagnosis. It communicates with Firebase for user authentication and stores diagnosis results and images.
4. **Firebase Subsystem:** Firebase handles user authentication, data storage, and database management. This subsystem supports the application by ensuring seamless data exchange and storage.

3.1 Transfer Learning

In our system, Transfer Learning plays a critical role by adapting the pre-trained MobileNetV2 model to specific datasets related to plant disease

diagnosis. The approach begins with two distinct open-source datasets: the PlantDis dataset and the Cassava dataset. These datasets undergo a thorough pre-processing phase, including labeling, data splitting (with an 80/20 train-test ratio), and rescaling to standardize the input. After pre-processing, the MobileNetV2 model is fine-tuned separately for each dataset by adjusting parameters such as learning rate and epoch count. The training process is designed to optimize the model's ability to generalize to new, unseen data by using techniques like batch normalization, dropout, and global average pooling. Once the training is complete, the models are evaluated on test datasets to ensure they meet the desired accuracy thresholds. The final models are then converted to TensorFlow Lite format, allowing them to be efficiently integrated into mobile applications. This approach not only accelerates the development process but also ensures that the models are robust and capable of accurate disease detection across different plant types.

3.2 flutter Application

The Flutter application serves as the front-end of our system, providing a user-friendly interface for interacting with the trained models. The app is designed with a modular architecture, consisting of multiple screens: Login, Register, Main, Result, and Settings pages. Each page is built using Flutter's widget-based system, allowing for efficient state management and smooth transitions between screens. The Main page is where users can upload images of plants for disease diagnosis. Upon selecting an image, the app invokes the relevant machine learning model, either PlantDis or Cassava, depending on the plant type selected by the user. The diagnosis results, along with any user feedback, are displayed on the Result page. The app is built to be responsive and scalable, ensuring that it performs well across various devices and screen sizes. By integrating the machine learning models directly into the app, we provide users with a powerful tool for real-time plant disease diagnosis, all within a sleek and intuitive interface.

3.3 Firebase

Firebase acts as the backbone of our application, handling all backend operations, including user authentication, data storage, and real-time database management. Firebase Authentication secures the app by managing user sign-ups, logins, and session persistence, ensuring that only authorized users can access the system. Firestore Database is employed to store user-specific data, such as plant disease diagnosis results and feedback, which are crucial for both user experience and model improvement. Firebase Storage is utilized to securely store the images uploaded by users for analysis, ensuring that the data is both accessible and protected. The integration of Firebase with our Flutter application allows for seamless data flow between the front-end and back-end, enabling real-time updates and efficient data retrieval. For instance, once a diagnosis is completed, the results are immediately saved to Firestore, and the corresponding images are uploaded to Firebase Storage. This setup not only ensures data integrity and security but also allows for scalability, making it easier to handle a growing user base and increasing amounts of data.

4. Opensource Dataset

4.1 Cassava_dataset

Overview

The Cassava dataset utilized in this project is derived from a model trained to classify a concatenation of the iNaturalist and the plants subset of ImageNet-21K called CropNet_dataset⁴. The plants subset comprises 1.57M images across 3775 classes, specifically focused on images with labels that have the synset plant.n.02 as a hypernym in the WordNet graph. In total, the concatenated dataset includes 2.15M images and 8864 classes. The dataset is particularly valuable for identifying four types of cassava plant diseases as well as one healthy condition, making it a crucial resource for this study.



Figure 8

⁴ Google CropNet Cassava Disease Classifier:

<https://www.kaggle.com/models/google/cropnet/tensorFlow2/classifier-cassava-disease-v1/1>

Structure of dataset

The **Cassava Leaf Disease Dataset** is organized into a structured format that facilitates the training of machine learning models for disease classification. The dataset primarily consists of a train folder, which is divided into five subfolders, each corresponding to a different disease class or a healthy state of the cassava leaf. The five subfolders within the train directory are:

1. **healthy**: Contains images of cassava leaves that are healthy and do not show signs of any disease.
2. **cmd**: Includes images of cassava leaves affected by Cassava Mosaic Disease (CMD).
3. **cgm**: Contains images of leaves affected by Cassava Green Mite (CGM).
4. **cbsd**: Holds images of leaves infected with Cassava Brown Streak Disease (CBSD).
5. **cbb**: Encompasses images of leaves suffering from Cassava Bacterial Blight (CBB).
















 healthy	 train-cbsd-0.jpg	Type: JPG File Dimensions: 625 x 500	Size: 142 KB
	 train-cbsd-1.jpg	Type: JPG File Dimensions: 625 x 500	Size: 151 KB
 cmd	 train-cbsd-2.jpg	Type: JPG File Dimensions: 625 x 500	Size: 138 KB
 cgm	 train-cbsd-3.jpg	Type: JPG File Dimensions: 625 x 500	Size: 130 KB
	 train-cbsd-4.jpg	Type: JPG File Dimensions: 500 x 666	Size: 46.9 KB
 cbsd	 train-cbsd-5.jpg	Type: JPG File Dimensions: 625 x 500	Size: 123 KB
	 train-cbsd-6.jpg	Type: JPG File Dimensions: 666 x 500	Size: 114 KB
 cbb	 train-cbsd-7.jpg	Type: JPG File Dimensions: 500 x 666	Size: 91.4 KB
	 train-cbsd-8.jpg	Type: JPG File Dimensions: 500 x 666	Size: 40.5 KB
	 train-cbsd-9.jpg	Type: JPG File Dimensions: 500 x 666	Size: 68.7 KB

Figure 9

Each of these subfolders is populated with images in JPEG format, with file names like train-cbsd-0.jpg, train-cbb-1.jpg, and so forth, indicating their respective disease categories. The images vary in dimensions and sizes, with most having a resolution of around 625 x 500 pixels. This standardized structure enables efficient organization and easy access to the images during the model training process. The dataset is highly organized, ensuring that each image is properly labeled according to its category, facilitating the accurate training of machine learning models to distinguish between different diseases and healthy leaves.

4.2 Plant_leave_diseases dataset

Overview

The dataset titled "Data for: Identification of Plant Leaf Diseases Using a 9-layer Deep Convolutional Neural Network"⁵ consists of 61,486 images across 39 different classes of plant leaves and background images. This comprehensive dataset was created to aid in the development and evaluation of machine learning models for plant disease identification. Various data augmentation techniques, including image flipping, gamma correction, noise injection, PCA color augmentation, rotation, and scaling, were applied to increase the dataset's size and variability, thereby enhancing the robustness of the trained models. This dataset provides a valuable resource for researchers aiming to improve the accuracy and efficiency of plant disease diagnosis using deep learning techniques.

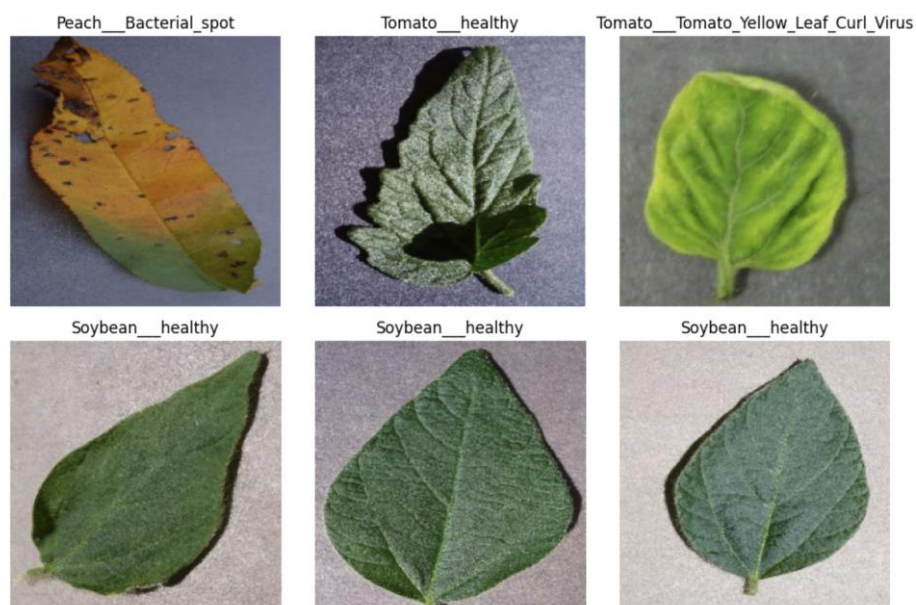


Figure 10

⁵ <https://data.mendeley.com/datasets/tywbtsjrjv/1>

Structure of dataset

The **PlantDis dataset** is organized into a well-structured format that facilitates the training of machine learning models for plant disease classification. This dataset consists of 39 different classes, each representing a specific plant disease or a healthy plant state, as well as background images. The dataset is comprehensive, containing 61,486 images, and each class is represented by a separate folder, named according to the plant disease or condition it contains.

Within each class folder, the images are stored in JPEG format, with file names like image (7).JPG, image (70).JPG, and so forth, indicating their respective sequences within the dataset. The images vary in dimensions and sizes, typically around 20 KB each, providing a balanced dataset in terms of image resolution and file size.

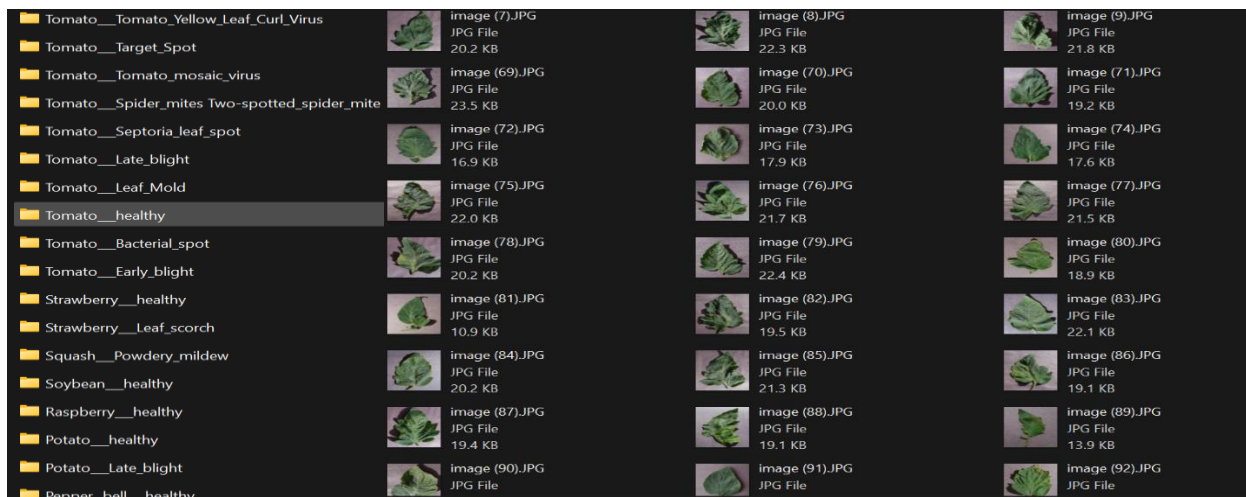


Figure 11

The dataset has already undergone significant augmentation by the original creators to enhance its size and diversity. Six different augmentation techniques were employed, including image flipping, Gamma correction, noise injection, PCA color augmentation, rotation, and scaling. These pre-applied augmentations ensure that the dataset is comprehensive and varied, allowing models trained on it to be robust and capable of generalizing well to new, unseen data. As a result, we did not need to perform any additional preprocessing, making the dataset ready for direct use in training our models.

5. Transfer learning

5.1 pre-Process

5.1.1 Cassava_dataset pre-process

Split based on the training

We used a method to efficiently load and preprocess our cassava dataset, which was crucial for managing the large volume of images. This method streamlined the process by automatically batching, shuffling, and applying necessary augmentations to the data.

Based on the original data split, we first divided the dataset into training and testing sets. Subsequently, the testing set was further split evenly into validation and testing sets, ensuring a robust data pipeline for our model training process. Fig 12 shows the final distribution of the dataset is as follows:

The result of the split:

labels	Description	Train	Val	Test
cmd	Mosaic Disease	1860	199	200
cbb	Bacterial Blight	326	35	35
cgm	Green Mite	541	58	58
cbsd	Brown Streak Disease	1010	108	109
healthy	Healthy	221	24	24

Figure 12

Pre-process of data

The primary purpose of applying data augmentation through ImageDataGenerator is to address the imbalance in the cassava dataset, where certain classes may have significantly more images than others. By augmenting the minority classes, we increase the likelihood that the model will learn robust features from all classes, reducing the bias toward the majority classes.

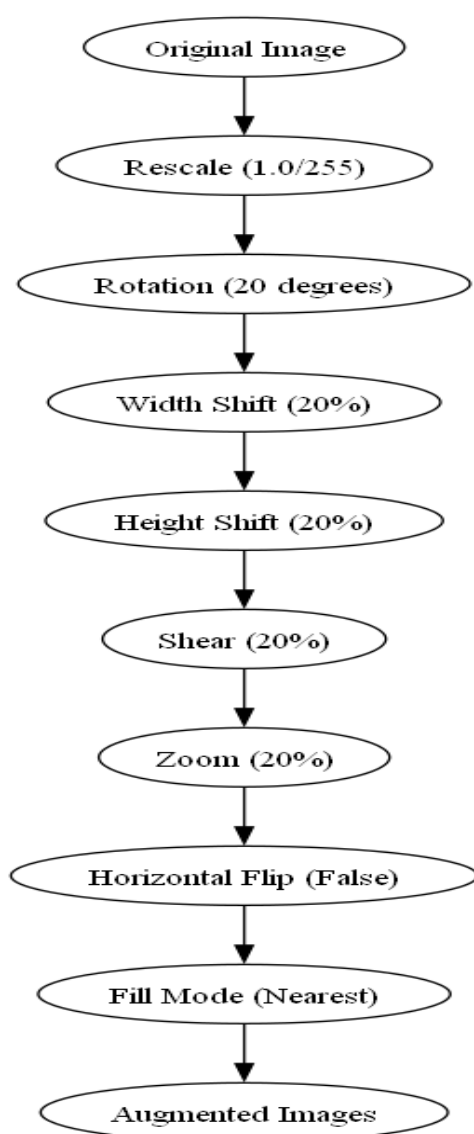


Figure 13

This technique helps in creating a more balanced dataset, leading to better model performance, particularly in terms of generalization to unseen data.

This strategic use of ImageDataGenerator ensures that our model is exposed to a wide variety of transformations, which is crucial for achieving high accuracy and robustness, especially when dealing with the inherent challenges of plant disease classification in the cassava dataset.

5.1.2 Plant_leave_diseases dataset pre-process

Split based on the training

Class	Train	Val	Test
Apple__Apple_scab	800	100	100
Apple__Black_rot	800	100	100
Apple__Cedar_apple_rust	800	100	100
Apple__healthy	1316	164	165
Background_without_leaves	914	114	115
Blueberry__healthy	1201	150	151
Cherry__Powdery_mildew	841	106	105
Cherry__healthy	800	100	100
Corn__Cercospora_leaf_spot Gray_leaf_spot	800	100	100
Corn__Common_rust	953	120	119
Corn__Northern_Leaf_Blight	800	100	100
Corn__healthy	929	116	117
Grape__Black_rot	944	118	118
Grape__Esca_(Black_Measles)	1106	139	138
Grape__Leaf_blight_(Isariopsis_Leaf_Spot)	860	108	108
Grape__healthy	800	100	100
Orange__Haunglongbing_(Citrus_greening)	4405	551	551
Peach__Bacterial_spot	1837	230	230
Peach__healthy	800	100	100
Pepper,_bell__Bacterial_spot	800	100	100
Pepper,_bell__healthy	1182	148	148

Potato__Early_blight	800	100	100
Potato__Late_blight	800	100	100
Potato__healthy	800	100	100
Raspberry__healthy	800	100	100
Soybean__healthy	4072	509	509
Squash__Powdery_mildew	1468	184	183
Strawberry__Leaf_scorch	887	111	111
Strawberry__healthy	800	100	100
Tomato__Bacterial_spot	1701	213	213
Tomato__Early_blight	800	100	100
Tomato__Late_blight	1527	191	191
Tomato__Leaf_Mold	800	100	100
Tomato__Septoria_leaf_spot	1416	177	178
Tomato__Spider_mites Two-spotted_spider_mite	1340	168	168
Tomato__Target_Spot	1123	141	140
Tomato__Tomato_Yellow_Leaf_Curl_Virus	4285	536	536
Tomato__Tomato_mosaic_virus	800	100	100
Tomato__healthy	1272	159	160

Figure 14

Pre-process of data

The plant dataset we used is an open-source dataset that has already undergone augmentation, which means it includes a variety of transformations such as rotations, flips, and color adjustments to help the model generalize better across different types of plant images.

Despite the inherent augmentation within the dataset, we implemented additional preprocessing steps to further refine the data and optimize it for training our machine learning models. These steps involve systematically organizing the data into classes, creating structured datasets, splitting them into training and validation sets, and mapping them into a format that is ready for model ingestion.

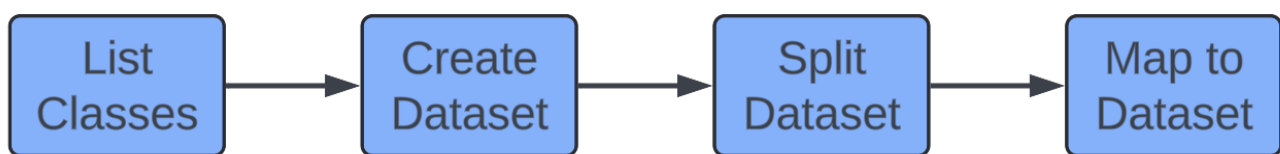


Figure 15

5.2 Training Process

In this project, we leveraged transfer learning to train two separate models, each designed for different plant disease diagnosis tasks. Transfer learning allows us to utilize a pre-trained model, in this case, MobileNetV2, which was initially trained on the ImageNet dataset, and fine-tune it on our specific datasets. This approach significantly reduces training time and improves model accuracy by building upon the existing knowledge of the pre-trained model. The two models we trained are:

1. **PlantDis Model:** Focused on a comprehensive plant disease dataset.



Figure 16

2. **CropNet Model:** Specifically designed for the cassava disease dataset.



Figure 17

5.2.1 Plantdis model

For the PlantDis model, we employed the MobileNetV2 architecture as the base model and modified it to suit our specific classification task. Here are the key details of the model:

Base Model:

- **Architecture:** MobileNetV2
- **Input Shape:** (224, 224, 3)
- **Pre-trained Weights:** ImageNet
- **Trainable Layers:** All layers of the base model were frozen to preserve the pre-trained weights.

```
Model: "model"
```

Layer (type)	Output Shape	Param #	Connected to
=====			
input_1 (InputLayer)	[(None, 224, 224, 3)]	0	[]
Conv1 (Conv2D)	(None, 112, 112, 32)	864	['input_1[0][0]']
bn_Conv1 (BatchNormalization)	(None, 112, 112, 32)	128	['Conv1[0][0]']
Conv1_relu (ReLU)	(None, 112, 112, 32)	0	['bn_Conv1[0][0]']
expanded_conv_depthwise (Depth wiseConv2D)	(None, 112, 112, 32)	288	['Conv1_relu[0][0]']
expanded_conv_depthwise_BN (Ba tchNormalization)	(None, 112, 112, 32)	128	['expanded_conv_depthwise[0][0]']
expanded_conv_depthwise_relu (ReLU)	(None, 112, 112, 32)	0	['expanded_conv_depthwise_BN[0][0]']
...			
Total params: 3,614,823			
Trainable params: 1,354,279			
Non-trainable params: 2,260,544			

Figure 18

Adding Custom Layers

In our model architecture, we have extended the base MobileNetV2 model by adding three new layers, and ultimately using a Softmax layer to output the possible 39 labels. The detailed architecture is as follows:

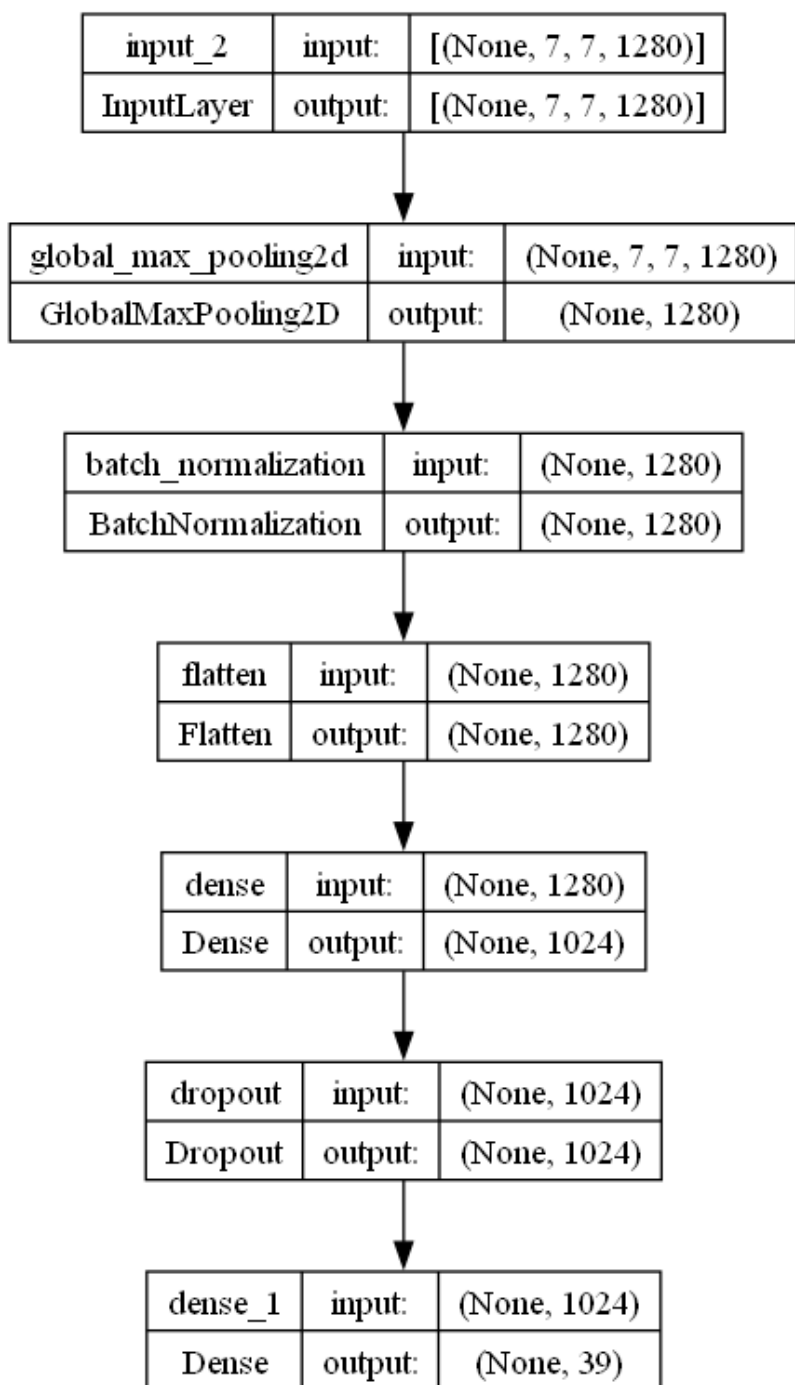


Figure 19

Custom Layers:

1. Global Max Pooling: `GlobalMaxPooling2D()`, used to downsample the feature maps from the base model.
2. Batch Normalization: Applied to normalize the output of the pooling layer.
3. Flatten Layer: `Flatten()` used to convert the pooled feature maps into a 1D vector.
4. Dense Layer: `Dense(1024, activation='relu')` with ReLU activation for additional feature extraction.
5. Dropout Layer: `Dropout(0.5)` applied for regularization to prevent overfitting.
6. Softmax activation:
The final dense layer with softmax activation outputs a probability distribution over the 39 classes. This layer is used for multi-class classification.

Model Compilation Choices:

During the model compilation step, we carefully selected the optimizer, loss function, and metrics to ensure that the model would perform efficiently and effectively.

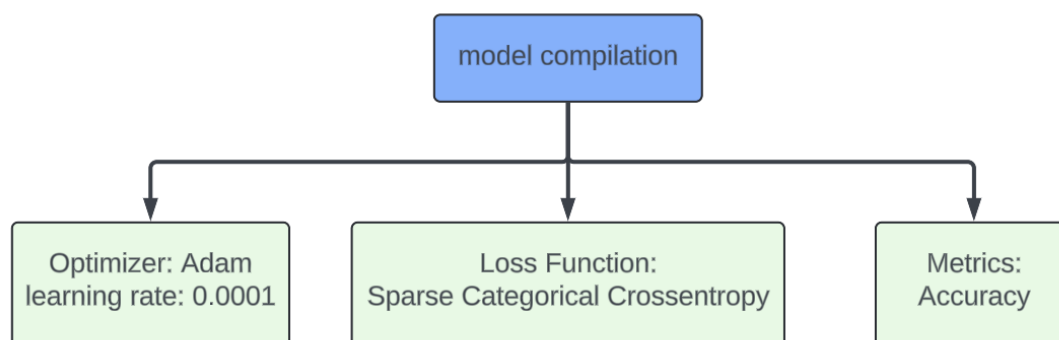


Figure 20

Optimizer: Adam

We chose the Adam optimizer⁶ (`tf.keras.optimizers.Adam`) for this model due to its adaptive learning rate capabilities, which allow it to converge faster and more reliably compared to other optimizers. Adam combines the advantages of two other extensions of stochastic gradient descent: Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp), making it particularly well-suited for complex models and large datasets. The learning rate was set to 0.0001, a conservative choice to prevent overshooting the optimal minima during training.

⁶ `tf.keras.optimizers.Adam`: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

Loss Function: Sparse Categorical Crossentropy

The loss function selected was `sparse_categorical_crossentropy`,⁷ which is appropriate for multi-class classification problems where the target variable is provided as integers (rather than one-hot encoded vectors). This function computes the cross-entropy loss between the true labels and the predicted labels, penalizing incorrect predictions more heavily, thus pushing the model towards higher accuracy.

Metrics: Accuracy

We chose accuracy⁸ as the metric to evaluate the model's performance. This metric calculates the ratio of correctly predicted instances to the total instances in the dataset, providing a straightforward measure of the model's ability to classify the input images correctly. Given that the primary goal is to maximize the number of correctly classified plant disease images, accuracy serves as a relevant and intuitive metric.

⁷ Sparse Categorical Crossentropy:

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

⁸ Metrics: Accuracy: https://keras.io/api/metrics/accuracy_metrics/

5.2.2 Cropnet model

The CropNet_model is designed to diagnose cassava plant diseases. We also utilized the MobileNetV2 architecture as the base model but with some variations in the fine-tuning process. Here are the details:

Base Model:

- **Architecture:** MobileNetV2
- **Input Shape:** (224, 224, 3)
- **Pre-trained Weights:** ImageNet
- **Trainable Layers:** Initially, all layers of the base model were frozen. However, fine-tuning was intended but not implemented in the final code version.

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_224 (Functional)	(None, 7, 7, 1280)	2257984
flatten (Flatten)	(None, 62720)	0
batch_normalization (Batch Normalization)	(None, 62720)	250880
dropout (Dropout)	(None, 62720)	0
dense (Dense)	(None, 5)	313605

```
=====  
Total params: 2,822,469  
Trainable params: 2,662,917  
Non-trainable params: 159,552  
=====
```

Figure 21

Data Augmentation:

- **Augmentation Techniques:** The input data was augmented with random flips, rotations, zooms, and contrast adjustments to improve the model's robustness.

Custom Layers:

- **Global Average Pooling:** GlobalAveragePooling2D(), used to reduce the feature maps to a single vector.
- **Batch Normalization:** Applied to stabilize and accelerate the training.
- **Dropout Layers:** Two Dropout(0.5) layers were added to reduce overfitting.
- **Output Layer:** Dense(5, activation='softmax') for classifying the five classes present in the cassava dataset.

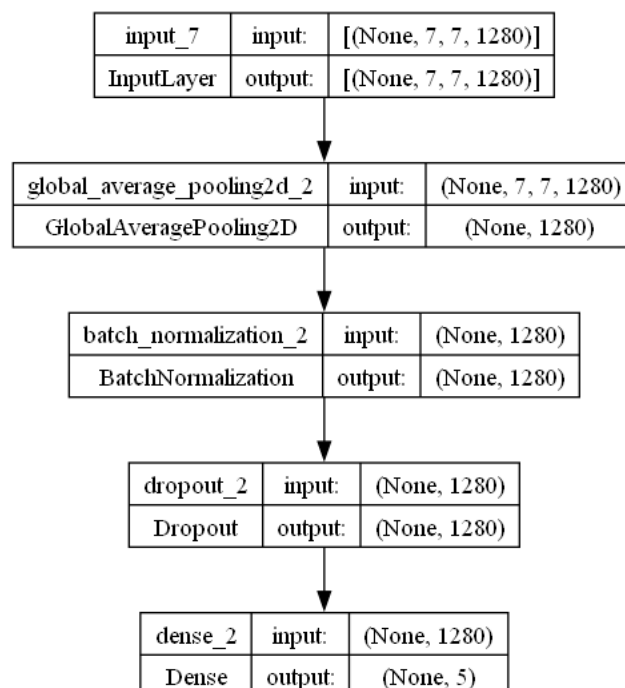


Figure 22

Model Compilation Choices:

In the **CropNet** model, the compilation choices were carefully selected to align with the goals of optimizing performance while maintaining accuracy across a diverse set of plant disease categories. Here's a detailed explanation of the choices made during the model compilation step:

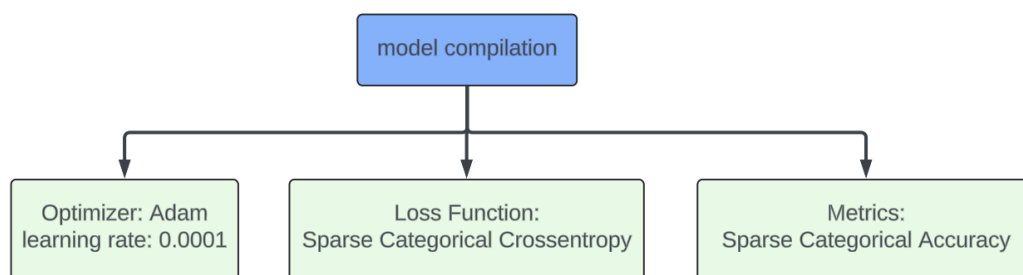


Figure 23

Optimizer:

- We used the **Adam** optimizer, which is known for its efficiency in handling large datasets and its ability to adapt the learning rate during training. Adam combines the advantages of two other extensions of stochastic gradient descent: adaptive gradient algorithm (AdaGrad) and root mean square propagation (RMSProp). This optimizer was chosen because it often yields faster convergence and better overall performance, particularly when working with complex neural networks like MobileNetV2.

- The **learning rate** was set to a **base_learning_rate** of 0.0001, a relatively low value that helps prevent the model from making large updates to the weights. This careful choice of learning rate allows for more stable and gradual convergence during training, which is crucial when fine-tuning pre-trained models like MobileNetV2.

Loss Function:

- **SparseCategoricalCrossentropy** was chosen due to its suitability for multi-class classification problems with integer labels, such as predicting various plant diseases. This loss function efficiently handles large-scale categorical data and optimizes the probability distribution over output classes, enhancing the model's predictive accuracy.

Metrics:

- **SparseCategoricalAccuracy** was selected to evaluate model performance. It provides a clear measure of how often the model's predictions match the correct labels. By tracking accuracy during training and validation, we can ensure the model is effectively learning and improving over time.

6. Application

As the fig 24 shows, the Flutter application is structured around five main pages: Login Page, Register Page, Main Page, Result Page, and Setting Page. These pages manage essential functions such as user authentication, plant disease diagnosis, and application settings. Each page is linked to a corresponding state class (`_LoginPageState`, `_RegisterPageState`, `_MyAppState`, `_MyImagePickerState`, `_ResultPageState`, and `_SettingPageState`) that handles the UI and user interactions. The workflow starts with user login or registration, continues with the selection and analysis of plant images, and ends with displaying the diagnostic results and collecting user feedback. The application's architecture ensures seamless transitions between these steps, providing users with an intuitive and efficient experience. The Result Page also allows users to submit feedback on the diagnosis, which is then stored in Firebase for further analysis and model improvement. The Setting Page enables customization of features like dark mode and text-to-speech, enhancing user experience.

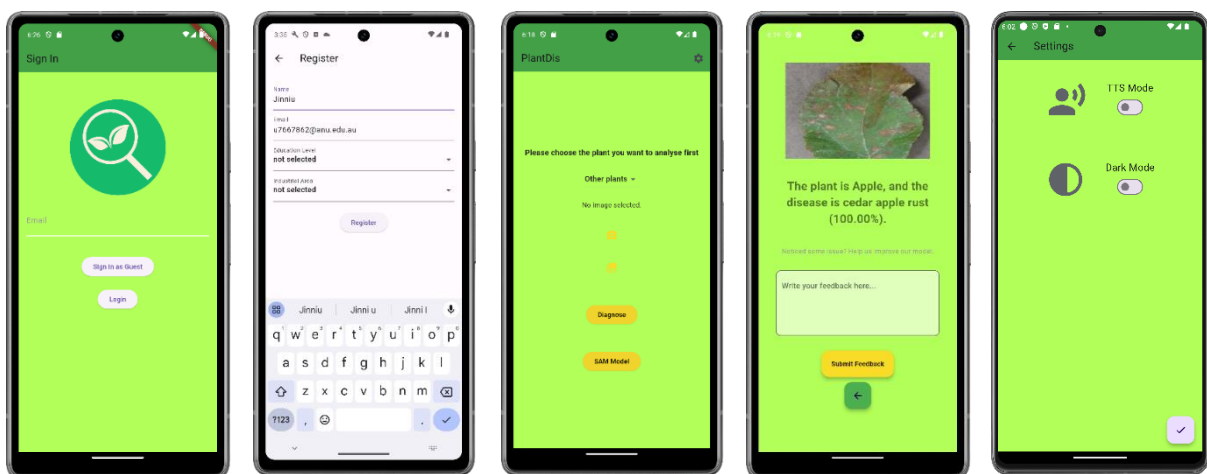


Figure 24

6.1 Login and Register Page

In the Flutter application, the login and register pages are integral to the user authentication process. If a user already has an account, they can directly log in via the LoginPage, which checks the user's credentials against Firebase Authentication. If the user is not registered, they can click the "Sign in as Guest" button, which navigates them to the RegisterPage. Here, they can enter their details, which are validated and stored in Firebase Authentication. Once registered, users can log in using their ANU email credentials on subsequent visits.

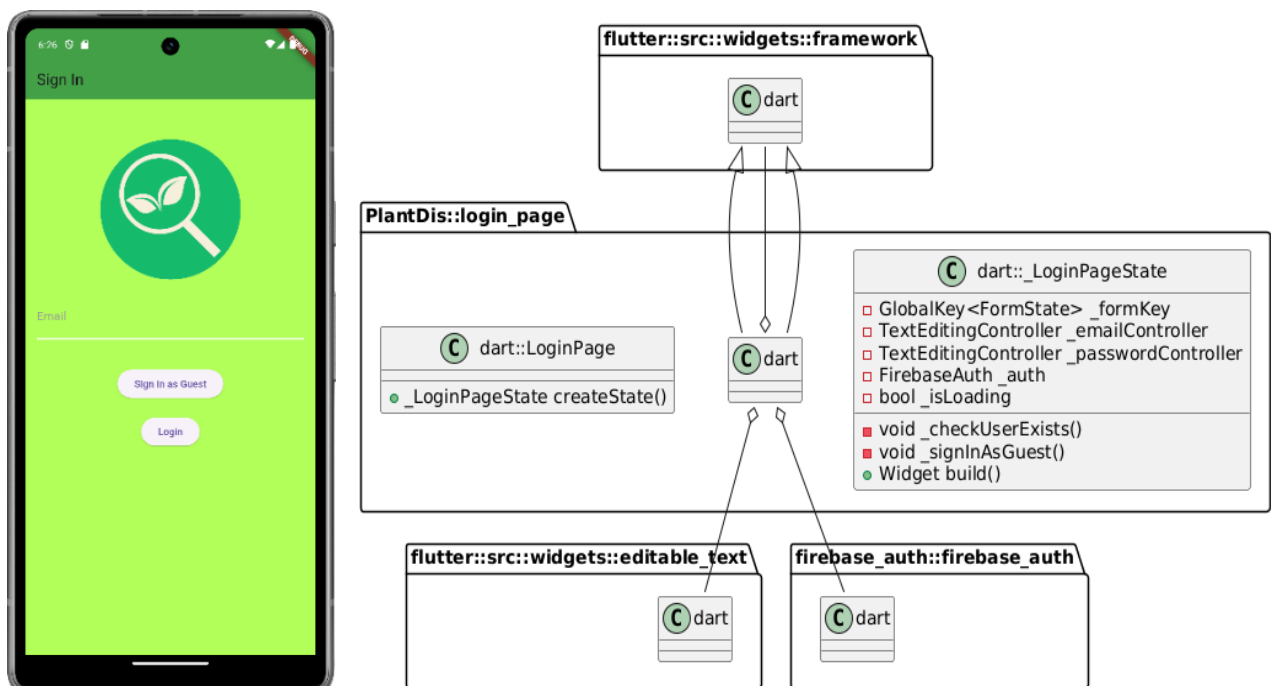


Figure 25

Login Page structure:

The **LoginPage** serves as the initial entry point for users, managing both authentication and user registration processes in a secure and user-friendly manner. When a user attempts to log in, the LoginPage utilizes a form that captures the user's email and password. These inputs are managed through `TextEditingController` instances, ensuring the data is correctly processed. Upon submission, the `_checkUserExists()` method is triggered. This method queries the Firebase Firestore database to verify if the email is associated with an existing account. If a match is found, the user is authenticated via Firebase's `signInWithEmailAndPassword()` method. Successful authentication redirects the user to the main application page using `Navigator.pushReplacement()`, ensuring the login page is no longer accessible, which helps maintain session security.

For users who are not registered, the "Sign in as Guest" option is available. Clicking this button triggers the `_signInAsGuest()` method, which navigates the user to the RegisterPage, allowing them to create a new account. This method ensures a seamless transition from guest access to full registration, guiding new users through the necessary steps to join the platform. Throughout the process, the app provides real-time feedback, ensuring users are informed of their progress or any errors, creating a smooth and intuitive user experience. This comprehensive approach ensures that the **LoginPage** effectively manages user access, balancing security with ease of use.

RegisterPage structure:

The registration process involves validating the user's inputs through a form and storing the new user's information in Firebase Authentication. Once registered, the user is redirected to the main application page, and future logins can be performed using the newly created ANU account.

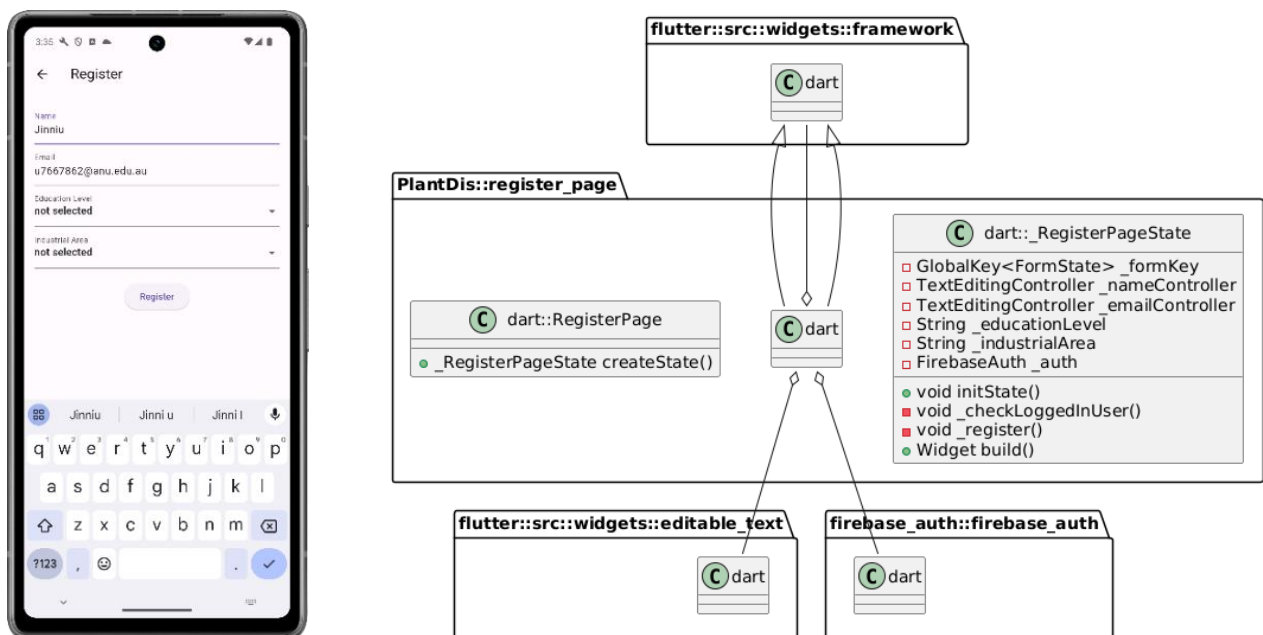


Figure 26

This process ensures that both new and existing users can easily navigate through the authentication process, providing a seamless transition between login and registration.

6.2 Main page

The Main page, also called Home Page of the PlantDis mobile application serves as the central hub for users. The MyApp class serves as the entry point of the application, initializing the app and navigating to the appropriate page based on the user's authentication state. The MyAppHome class, associated with the MyAppState, manages the primary logic, including the toggle for dark mode and communication with the Firebase backend through the `_saveResultToFirestore` method. The MyImagePickerState class is central to handling the user's image selection and plant disease diagnosis. It interacts with machine learning models such as CropCassavaModel and PlantVillageModel to process images and generate predictions. Additionally, the MyImagePickerStateTTS class manages the Text-to-Speech functionality, enhancing accessibility by providing auditory feedback on diagnosis results.

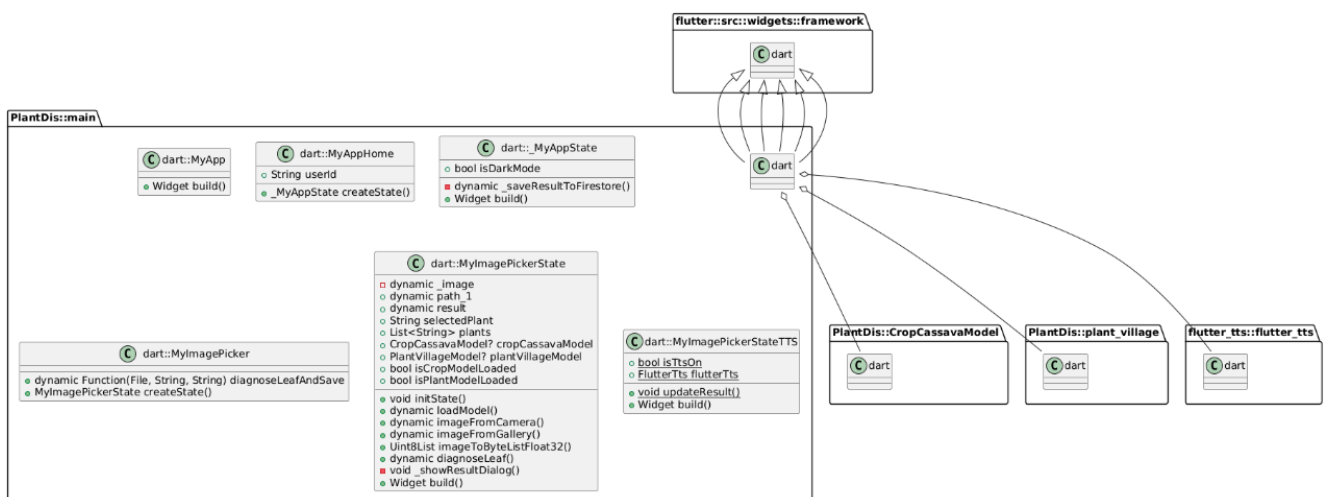


Figure 27

It features a user-friendly interface with a dropdown list allowing users to select the plant type they wish to analyse. This selection will load different models. The cassava options will load the Cropnet_model and the other plants will load the PlantDis model.

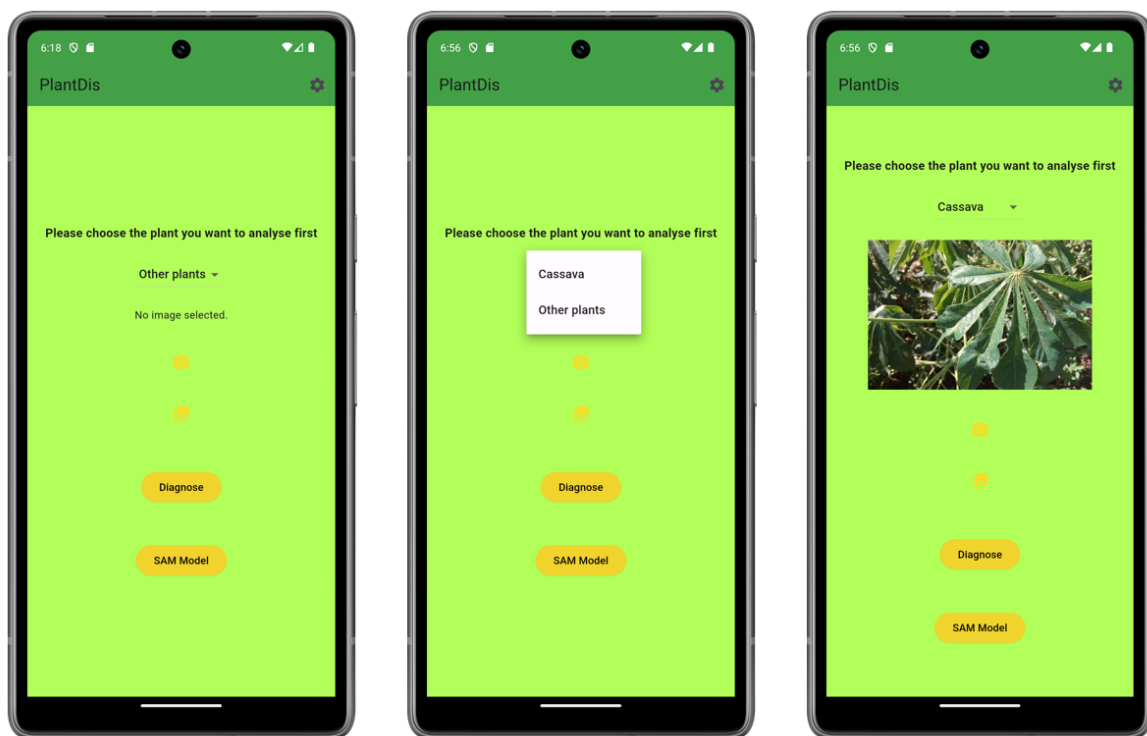


Figure 28

Beneath the dropdown, options are available for image selection either via camera or gallery. Key actions include the "Diagnose" button to initiate plant disease analysis.

About the **Image Upload and Diagnosis button**, users can upload or capture an image of the plant leaf for analysis. Upon selection, the image is displayed, and users can press the "Diagnose" button to start the disease diagnosis process using the pre-loaded model. The result is then displayed or narrated, depending on user settings, providing a detailed analysis of the plant's health status.

6.3 Result Page

The **Result Page** in the PlantDis application displays the diagnosis results of the uploaded plant image, including the identified plant type, disease, and confidence level. It contains properties like image, result, and a function `saveResultToFirestore`, which are passed to the state. The state class, `_ResultPageState`, manages the page's UI and behavior, including handling feedback submission and navigation. The diagram also shows dependencies on Flutter's widget framework and interaction with the `TextEditingController` for managing user input in the feedback form.

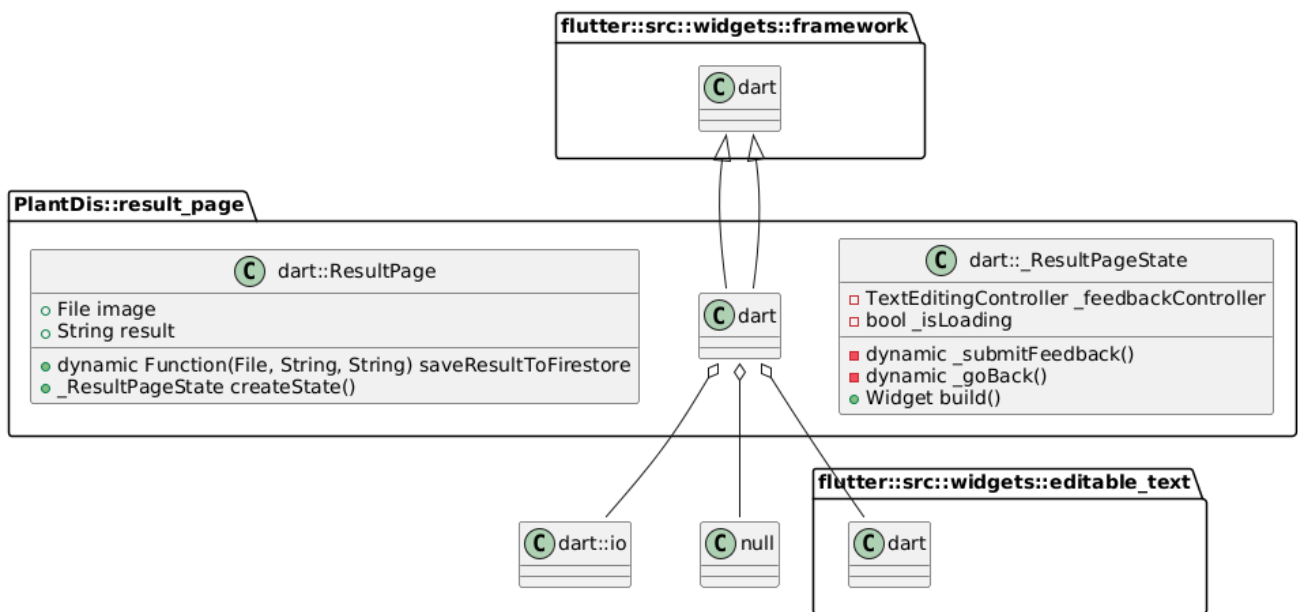


Figure 29

Both methods interact with the Database backend to ensure that user interactions (submitting feedback or returning to the main page) are correctly processed and stored.

It also provides users with options to submit feedback or return to the home page. Both the "Return" button and the "Submit Feedback" button will navigate the user back to the main page.

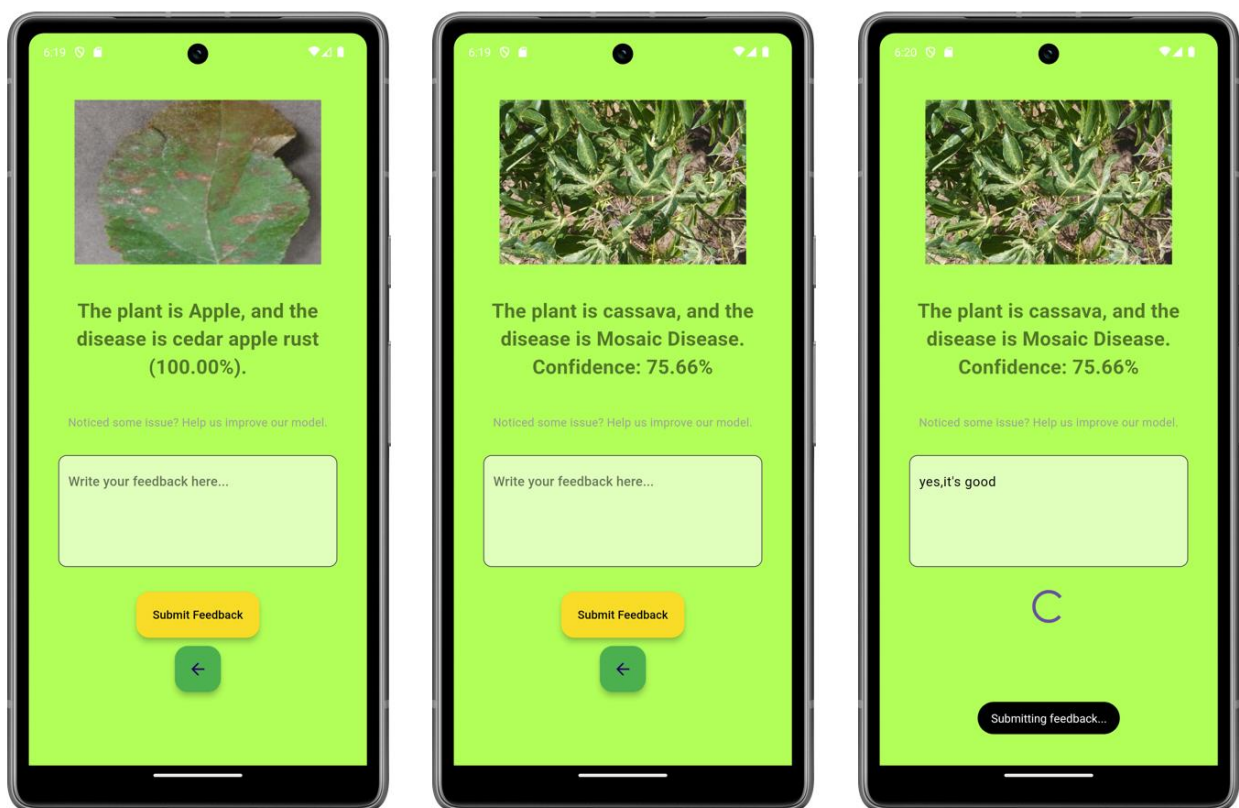


Figure 30

However, the "Submit Feedback" button requires the user to enter feedback in the provided text field before proceeding. This feedback, along with the image and diagnostic result, is then uploaded to the database. The feedback submission process includes displaying a loading indicator while the feedback is being sent to the server.

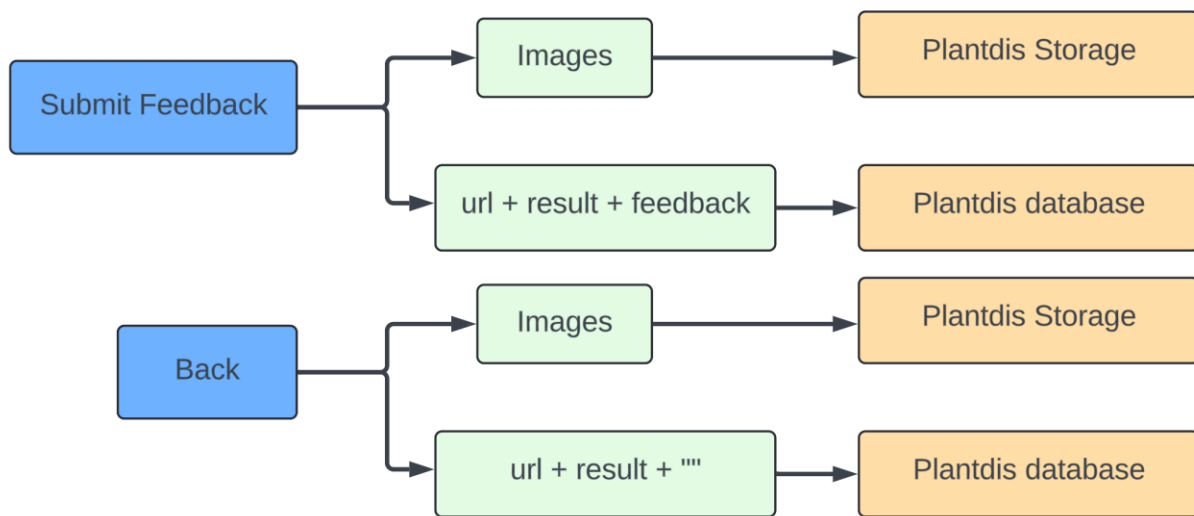


Figure 31

Displaying the Diagnosis Result

The diagnosis result is displayed in a bold, large font, stating the plant type, disease, and confidence percentage. The confidence is calculated based on the model's output probability and formatted to two decimal places.

Submission

In the **Result Page**, users can submit feedback by entering their comments in the provided text field and pressing the "Submit Feedback" button. The process, illustrated in the diagram, involves uploading the image to Plantdis Storage and storing the image URL, diagnosis result, and feedback in the Plantdis database. Alternatively, users can press the back button to save the result without providing feedback. During either action, a loading spinner is displayed, indicating that the feedback or result is being saved. Upon completion, a toast notification confirms the successful save, and the user is redirected to the home page.

6.4 Settings Page

The `SettingsPage` in the application allows users to toggle between Text-to-Speech (TTS) mode and Dark Mode. The class `SettingsPage` contains two boolean variables, `isTtsOn` and `isDarkMode`, which indicate whether these features are enabled. The `SettingsPageState` class manages the state of these settings, with methods like `_toggleTts()` and `_toggleDarkMode()` to update the boolean values when the user interacts with the toggles. The `initState()` method initializes these settings, and the `build()` method renders the UI based

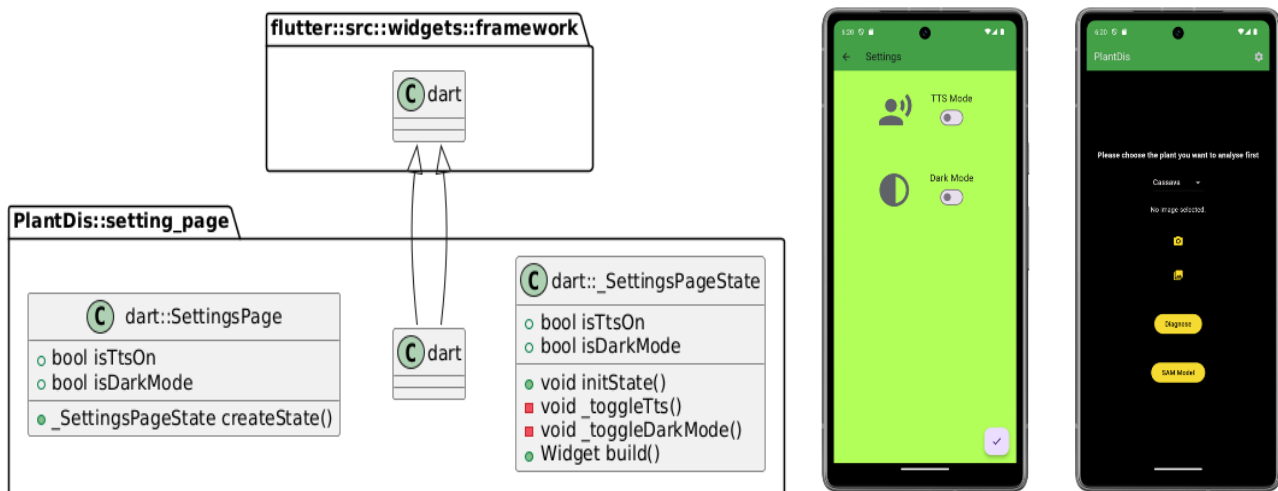


Figure 32

on the current state.

These settings are adjustable via toggle switches, providing users with a personalized experience.

6.4.1. Dark Mode

Dark Mode changes the app's visual theme to a darker color scheme, reducing eye strain and improving visibility in low-light conditions.

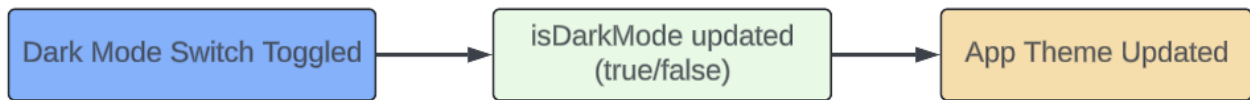


Figure 33

Users can toggle the isDarkMode boolean value, instantly switching the app's appearance between light and dark themes.

6.4.2. Text-to-Speech (TTS) Mode

When activated, the TTS mode reads out the diagnosis results aloud. This feature is particularly useful for users who prefer auditory feedback or have visual impairments.

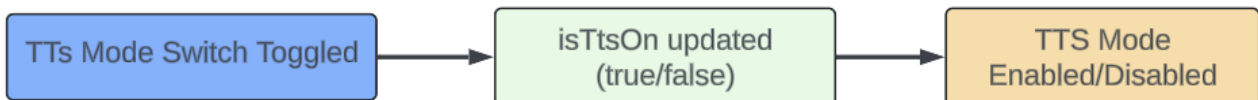


Figure 34

A switch toggles the isTtsOn boolean value, enabling or disabling the TTS functionality within the app.

6.5 Database

For the PlantDis application, we utilize Firebase as the backend database, referred to as the PlantDis Database, to handle several critical functions:

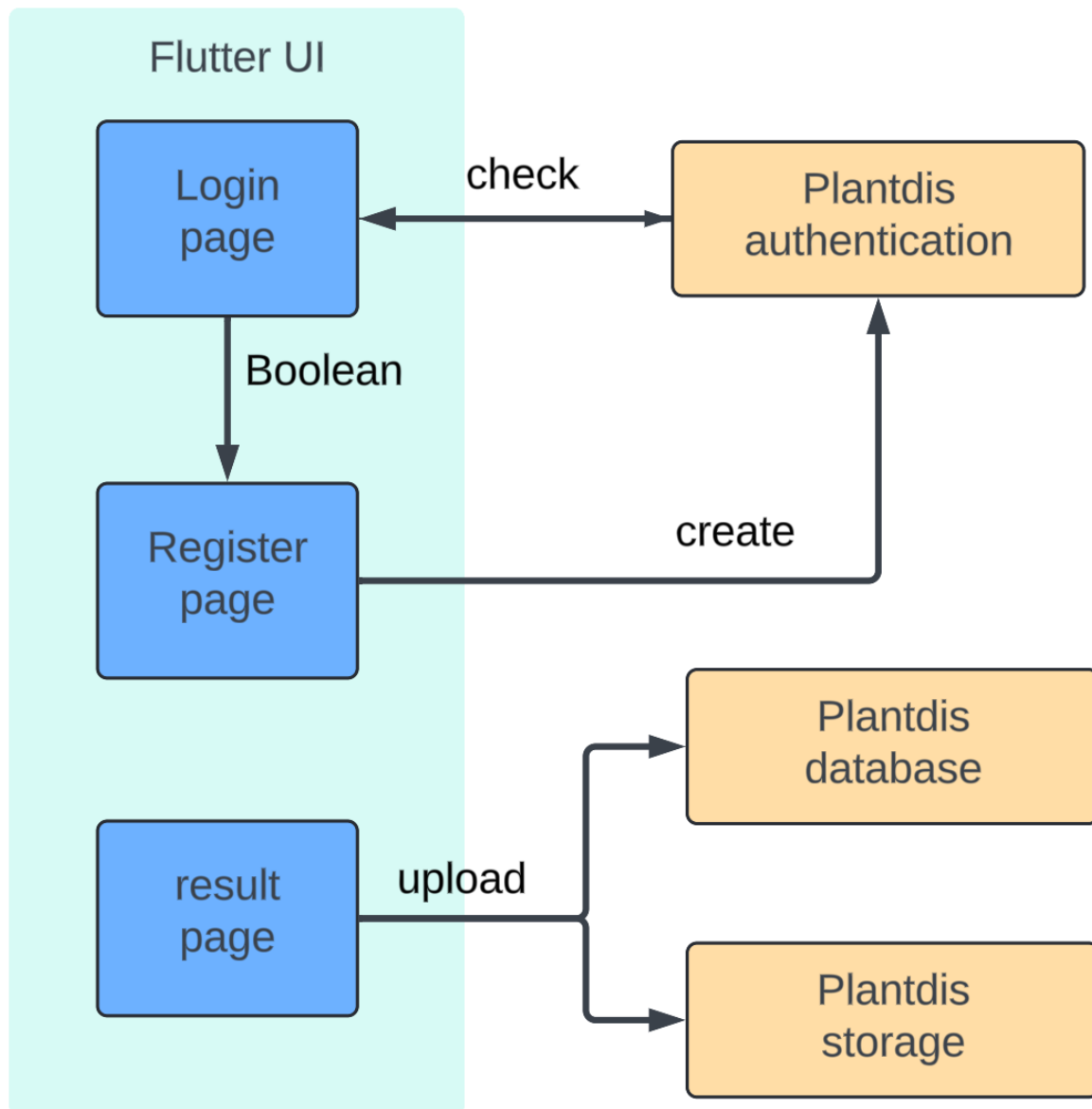


Figure 35

1. User Authentication with Firebase Auth

The PlantDis Database leverages Firebase Authentication (Firebase Auth) to manage user sign-in, sign-up, and authentication within the application.

This feature enables users to securely create accounts and log in, ensuring that access to the application is protected. Our database supports various authentication methods, including email/password and Google Sign-In, providing a flexible and secure way for users to authenticate themselves.

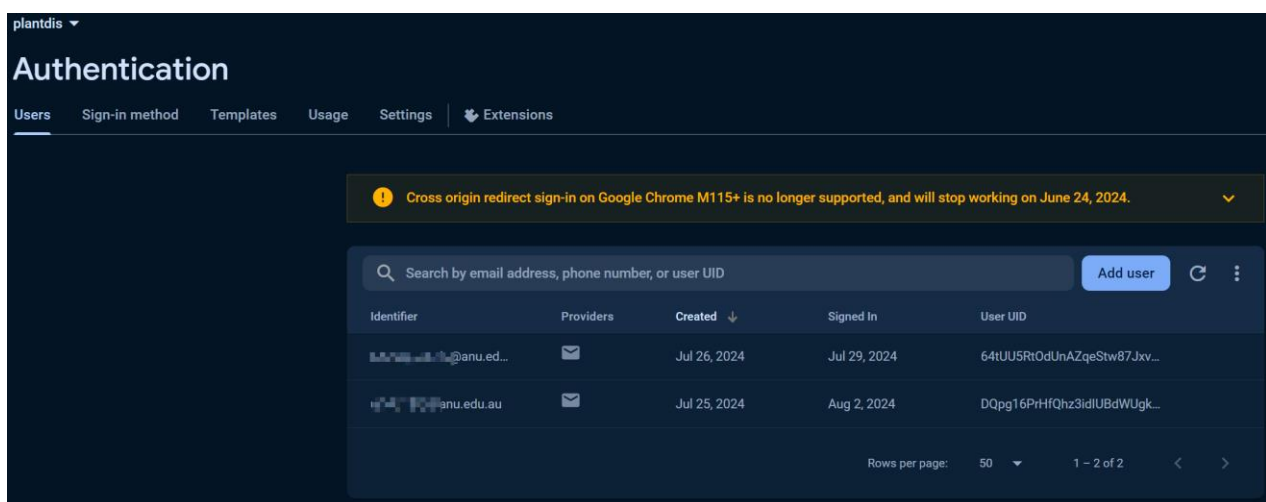


Figure 36

2. Image Storage with Firebase Storage

The PlantDis Database also uses Firebase Storage to manage the storage of images captured or uploaded by users. When users submit an image of a plant leaf for disease diagnosis, the image is securely stored in the PlantDis Database. Firebase Storage, optimized for handling large files like images and videos, facilitates secure file uploads and downloads directly from the client-side Flutter app, ensuring that user data is safely managed and readily accessible.

Test

The testing process is divided into two parts: **Testing with Python** and **Testing with the App**.

- **Testing with Python:** This segment details the initial evaluation of the trained models within the Python environment, focusing on performance metrics such as accuracy, loss, and validation results. These metrics provide insights into how well the models generalize on unseen data after the transfer learning process.
- **Testing with the App:** This part covers the real-world application of the models within the mobile app. It includes performance assessments conducted through the app's result page, where users can input images and receive predictions. This testing phase ensures that the models are not only accurate but also efficient and user-friendly when deployed in a live environment.

7. Testing

7.1 Testing with python

7.1.1 PlantDis Model result

Validation Accuracy:

During the training of the model, the validation accuracy and loss were monitored to evaluate the model's performance on unseen data. After completing 50 epochs, the model achieved a remarkable validation accuracy of approximately **99.38%** on the validation dataset, as shown in the graph. The training process involved a gradual increase in accuracy and a consistent decrease in loss for both the training and validation datasets.

```
test_loss, test_accuracy = model.evaluate(val_ds)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_accuracy}')
✓ 8.2s
```



```
385/385 [=====] - 8s 21ms/step - loss: 0.0266 - accuracy: 0.9938
Test Loss: 0.026627330109477043
Test Accuracy: 0.9938196539878845
```

Figure 37

The graph depicting **Training and Validation Accuracy** shows that the accuracy quickly improves in the initial epochs and then stabilizes as it approaches its peak.

Similarly, the **Training and Validation Loss** graph demonstrates a rapid decrease in loss in the early epochs, which then plateaus as the model converges.

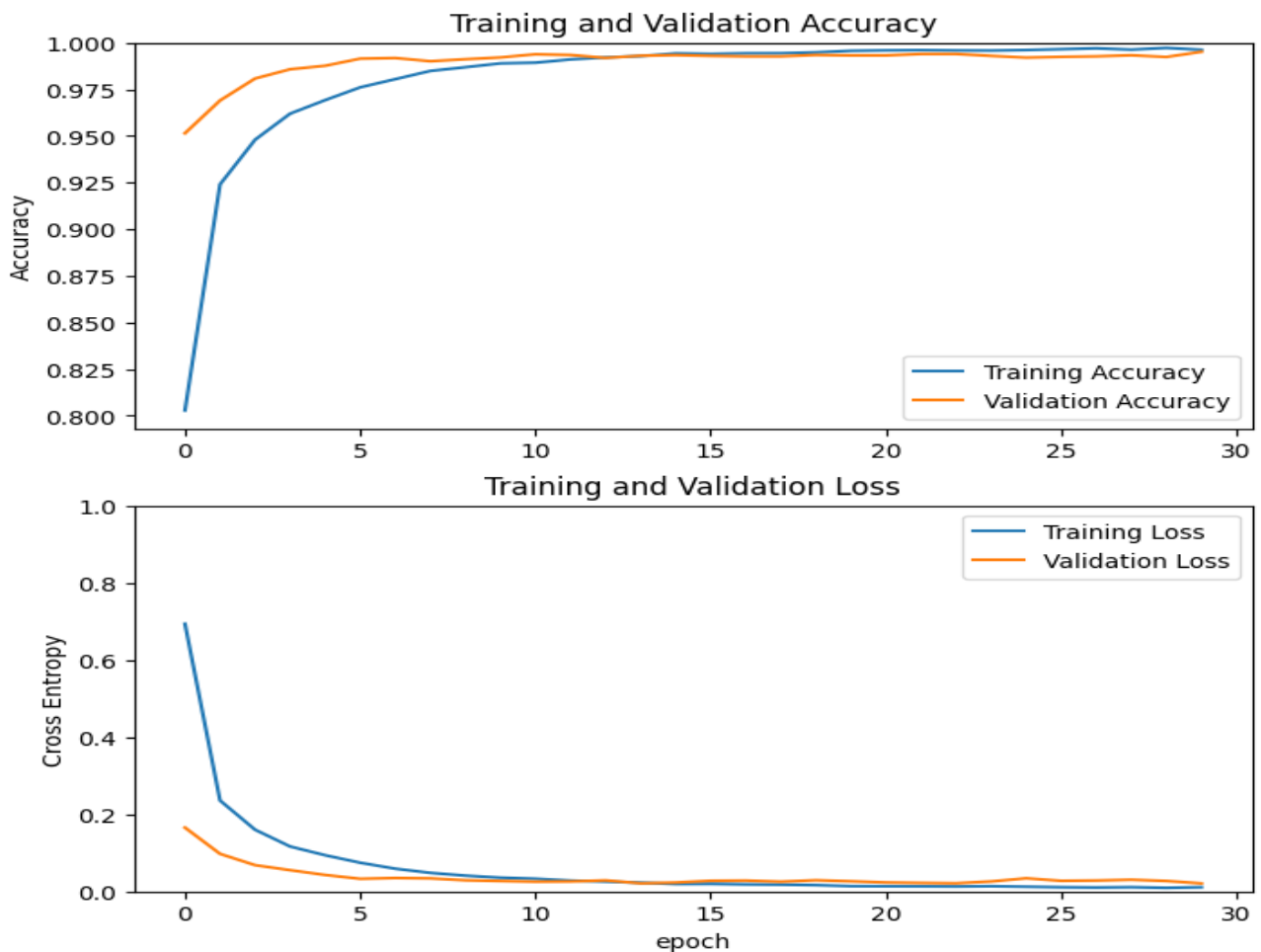


Figure 38

After converting the trained model to TensorFlow Lite (TFLite) format for deployment on mobile devices, the model was evaluated on the test dataset to assess its performance across different plant disease classes. The accuracy of the model across these classes was remarkable, with most classes achieving near-perfect accuracy.

Key Observations:

- **High Accuracy in Multiple Classes:** The model exhibited a high level of accuracy across the majority of classes, including Apple__Cedar_apple_rust, Apple__healthy, Background_without_leaves, and Blueberry__healthy, all achieving **100% accuracy**. This indicates that the model is highly reliable for diagnosing these specific plant diseases and conditions.
- **Variability in Performance:** While the model performed exceptionally well overall, there was some variability in accuracy across different classes. For example, the class Tomato__Early_blight had an accuracy of **83%**, and Tomato__Spider_mites Two-spotted_spider_mite achieved **88%** accuracy, which is lower compared to other classes. This suggests that while the model is robust, there may still be room for improvement in recognizing certain conditions, particularly where the dataset may have been less balanced or more complex.
- **Consistent Accuracy:** The model consistently achieved high accuracy for most plant diseases, reflecting its strong generalization capabilities after the conversion to TFLite. The average accuracy across all classes was **97.62%**, which demonstrates that the conversion process did not significantly impact the model's predictive performance.

Summary:

The TFLite model's average accuracy of **97.62%** suggests that it is well-suited for deployment in a real-time mobile application like PlantDis, ensuring that users receive accurate and reliable plant disease diagnoses directly on their devices. This high level of accuracy across various plant species and diseases highlights the model's effectiveness and the success of the transfer learning approach used during training.

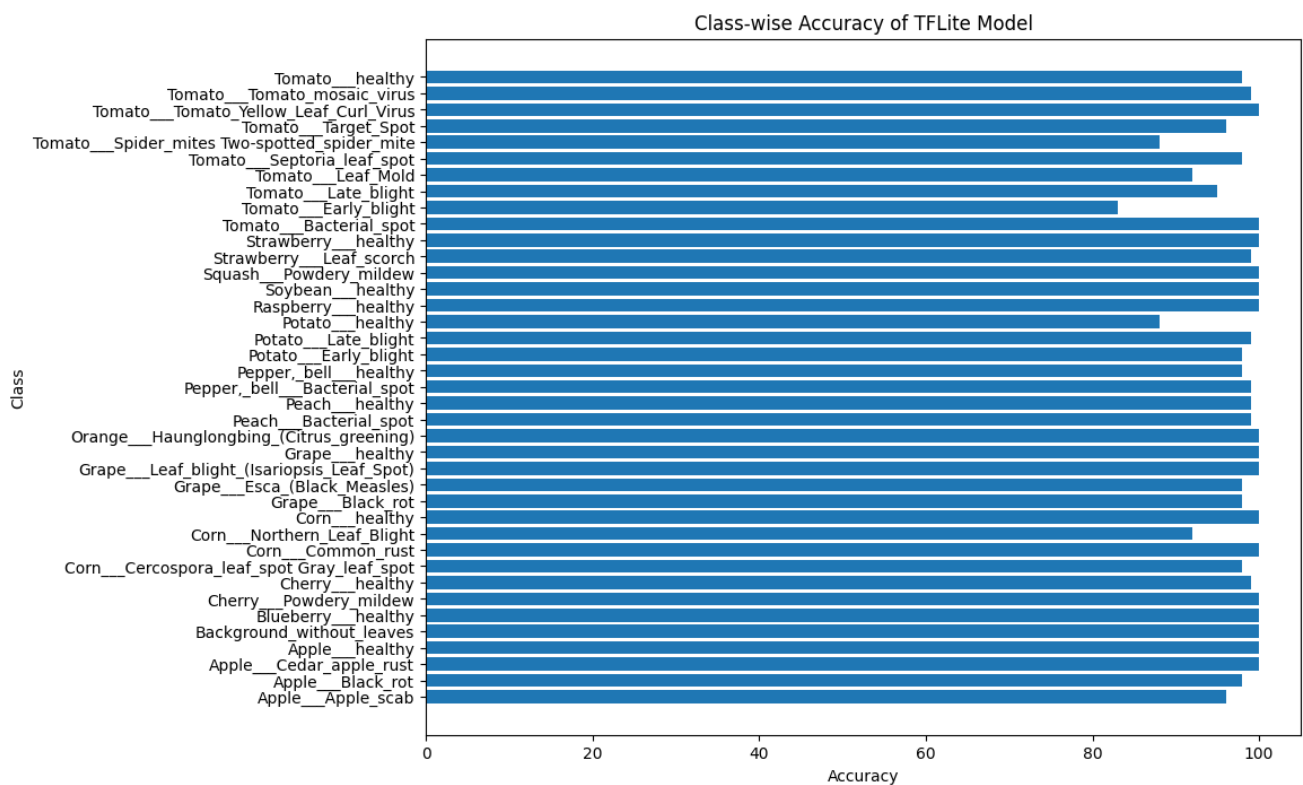


Figure 39

Performance on Unseen Data

We tested the generalization capabilities of our plant disease classification model by using images that were not part of the original training dataset, including completely different types of images such as bananas. The results were mixed: while the model performed well on some images that were within its trained categories, it struggled with images that were not represented in the training data. For instance, the model misclassified banana images as tomato or corn-related categories, which indicates a limitation in its ability to handle unseen objects.

File	Predicted Label	Comment
self_banana_1.jpeg	Tomato_healthy	Misclassified, as banana is not in the training set
self_banana_2.jpeg	Corn_healthy	Misclassified, as banana is not in the training set
self_corn(maize)(1).jpeg	Corn_common_rust	Correct Classification
self_corn(maize).jpeg	Corn_common_rust	Correct Classification
self_potato.jpeg	Pepper_healthy	Misclassified
self_tomato_2.jpeg	Corn_common_rust	Misclassified
self_tomato_3.jpeg	Apple_scab	Misclassified
self_tomato_4.jpeg	Tomato_bacterial_spot	Correct Classification

Figure 40

7.1.2 CropNet Model result

Validation Accuracy:

he results obtained from training the CropNet_model highlight several key aspects related to the dataset's inherent imbalance and the model's performance. As seen in the graphs, the model shows steady improvement in both training and validation accuracy over 50 epochs, though the validation accuracy plateaus around 70–75%. The training loss decreases consistently, but the validation loss exhibits fluctuations, indicating that the model is sensitive to the variations in the validation dataset.

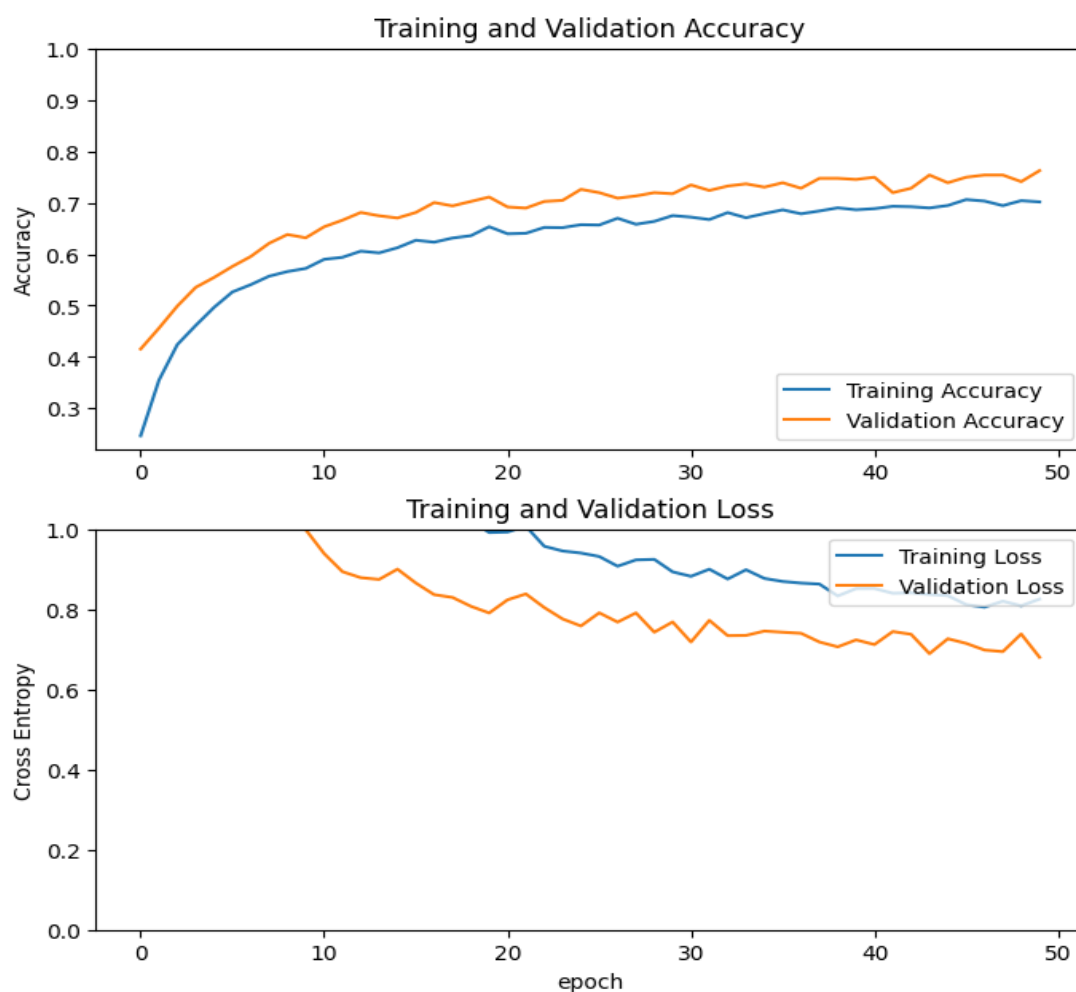


Figure 41

Given that the official cassava dataset is not balanced, with some classes having more examples than others, the model's performance reflects this challenge. For instance, while the model performs well on classes like "Mosaic Disease" and "Healthy," with accuracies around 88.8% and 88.5%, respectively, it struggles with "Green Mite," achieving only 54.8% accuracy. This discrepancy underscores the importance of data balance in training robust models capable of generalizing well to less-represented classes.

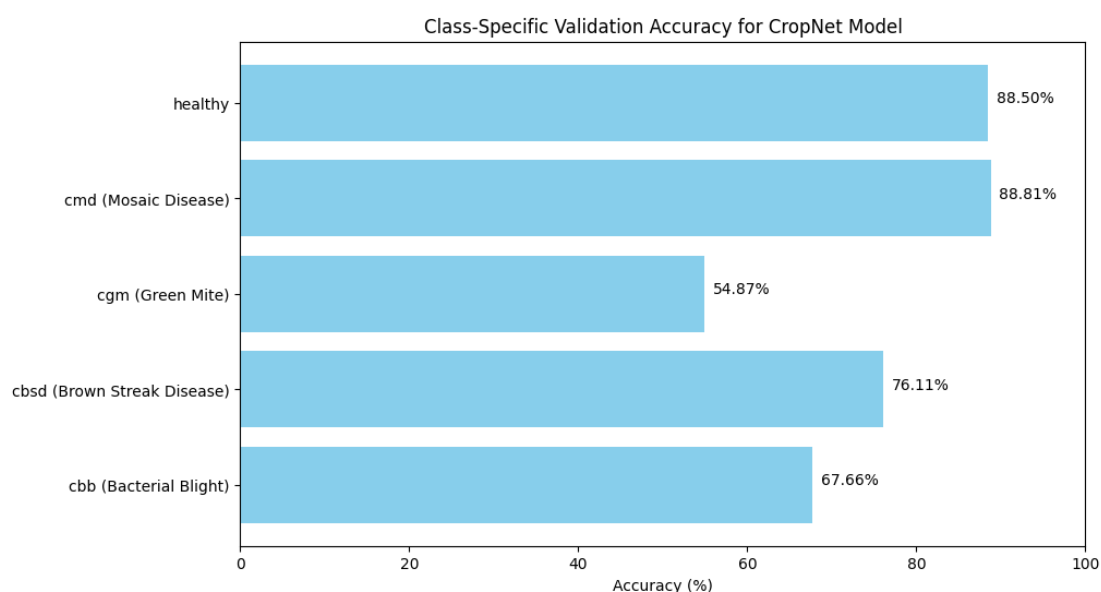


Figure 42

These results suggest that while the model is effective for certain well-represented classes, there is a need for further improvement in handling underrepresented classes. Techniques such as additional data augmentation, class weighting, or even collecting more data for the minority classes could be considered in future work to enhance the model's overall performance and generalization capability.

Performance on Unseen Data:

Given that the dataset used for training the CropNet_model consists solely of cassava leaf images, the model is inherently specialized for recognizing cassava leaf diseases. As a result, when presented with images of leaves from other plant species, the model is unable to accurately classify them and tends to categorize these unfamiliar leaves as "unknown" or incorrectly assign them to one of the known cassava disease classes. This limitation highlights the need for a broader and more diverse dataset if the goal is to create a more generalized model capable of identifying diseases across multiple plant species. Consequently, in our application, we employ a combination of two models to achieve a more comprehensive plant disease diagnosis, leveraging each model's strengths to cover different plant species and their associated conditions.

7.2 Testing with App

Model Output and Confidence Display

In the deployment of our trained model within the app, we made a strategic decision to present the confidence scores directly, without applying any mean filtering. This approach was chosen to provide a more transparent view of the model's predictions, displaying the confidence level for each classification as a percentage.

```
// Running the model on the preprocessed image
var output = await Tflite.runModelOnImage(
  path: path,
  numResults: 1, // Getting the top result
  imageMean: 0, // Mean normalization value
  imageStd: 255, // Standard deviation normalization value
);

// Checking the output and returning the predicted label with confidence
if (output != null && output.isNotEmpty) {
  String predictedLabel = output[0]['label'];
  double confidence = output[0]['confidence'] * 100; // Convert to percentage
  return ['$predictedLabel (${confidence.toStringAsFixed(2)}%)'];
} else {
  return ['un_known'];
}
}
```

Figure 43

In this implementation, the confidence score is calculated as a direct output from the model and multiplied by 100 to convert it into a percentage format. This raw confidence score is then displayed alongside the predicted label, giving users immediate insight into the model's certainty for each prediction.

Rationale: Displaying the confidence scores without additional filtering or adjustment serves a dual purpose. Firstly, it provides end-users with a clearer understanding of how certain the model is about its predictions, potentially

guiding their decisions when using the app. Secondly, for developers and researchers, these raw confidence scores offer critical data points that can be analyzed in subsequent iterations of model training. By examining cases where the confidence is low or where the model's predictions are incorrect, we can identify areas for improvement and refine the model to achieve higher accuracy in future versions.

Self-testing:

Using our app, we can easily obtain classification results for plant images, Fig

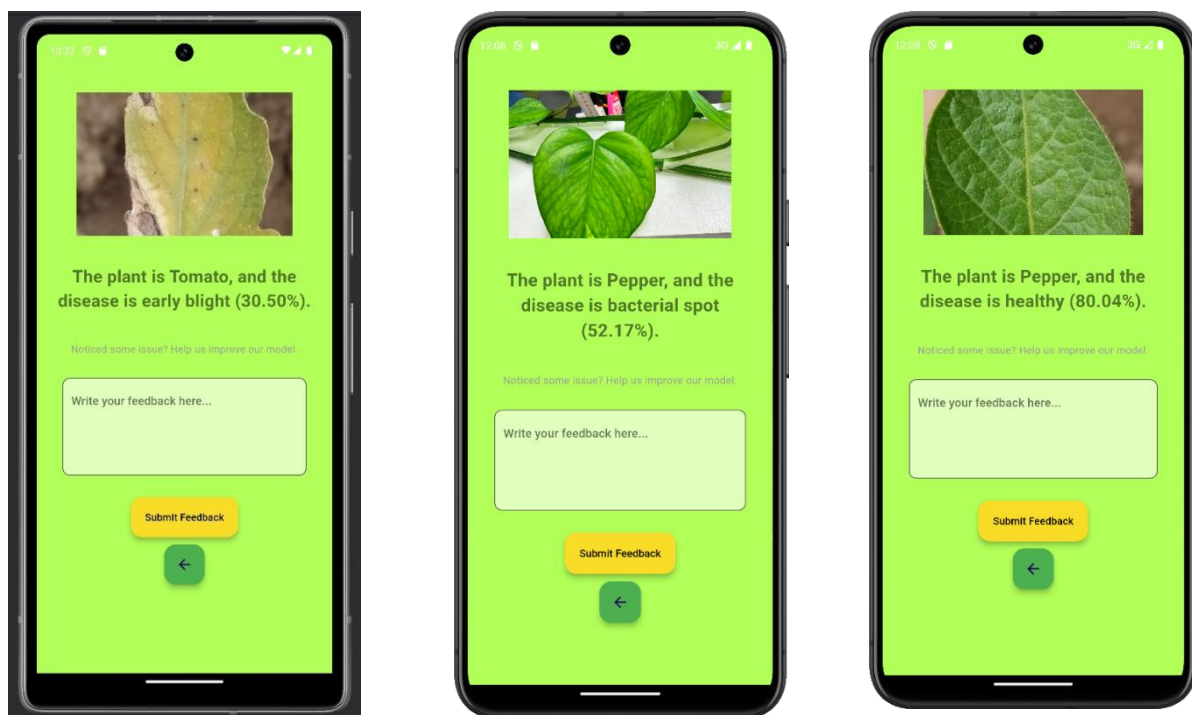


Figure 44

44 shows after complete with confidence scores that assist in pinpointing potential issues. The three screenshots below showcase images that were either captured in real-time or sourced from external websites, entirely different from the original dataset. As demonstrated by the confidence scores provided, there is still room for improvement in the model's accuracy. These

findings will be crucial as we continue to work on enhancing the model's generalization capabilities in future iterations.

8. Future Work

8.1 Integration of Segmentation Models

As part of the future work for the PlantDis application, we plan to integrate segmentation models like DeepLab to enhance the image analysis capabilities. After loading an image, users can click the SAM button to access a new page where they can use the segmentation model to generate masks for individual leaves. This allows for detailed analysis even when multiple leaves are present in the image. The DeepLab model, known for its state-of-the-art performance in semantic segmentation, will be employed to ensure accurate leaf segmentation. This feature is crucial for cases where different leaves exhibit varying disease symptoms, enabling more precise diagnosis and further research into plant health. Additionally, DeepLab's⁹ TFLite format will be used to ensure smooth integration into the mobile application.

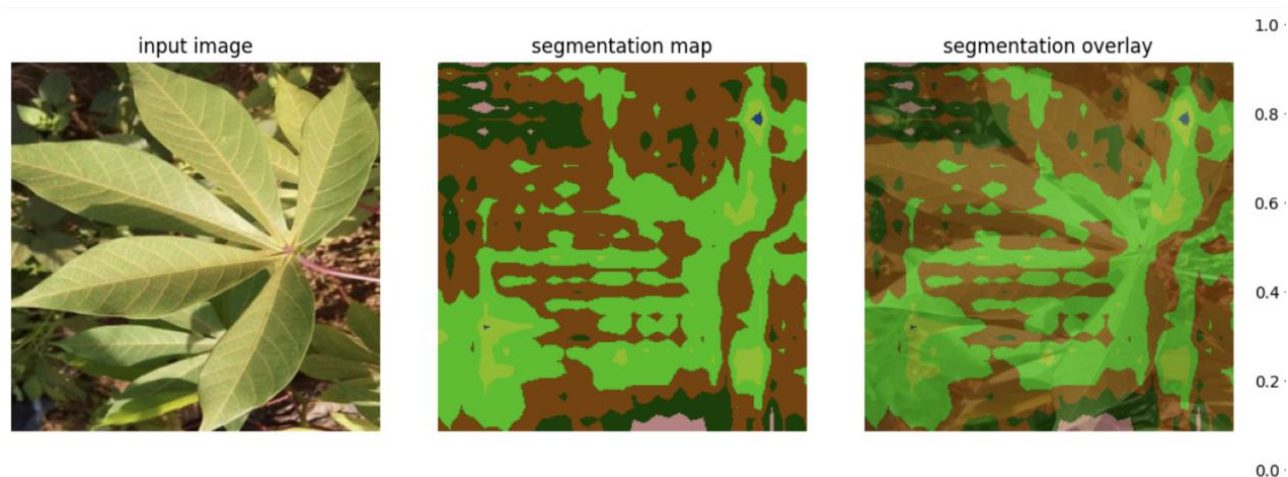


Figure 45

⁹ Deep Labelling for Semantic Image Segmentation:
<https://github.com/tensorflow/models/tree/master/research/deeplab>

8.2 Improving Model Generalization

To improve the generalization of the models in PlantDis, efforts will focus on both the dataset and the model itself:

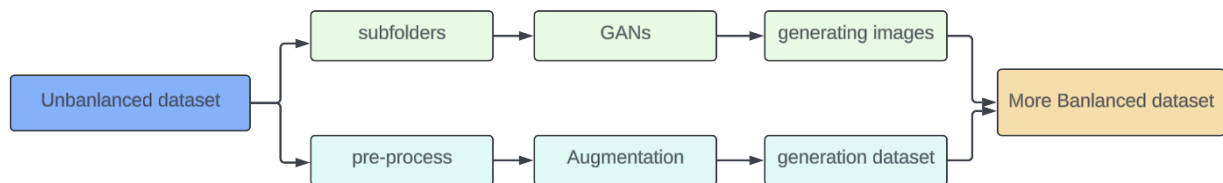


Figure 46

8.2.1 Dataset Expansion:

Balancing the Dataset:

Using GAN (Generative Adversarial Networks) to generate synthetic images for underrepresented classes can help balance the dataset, leading to better model generalization.

Data Augmentation:

Implementing auto-augmentation techniques or using ``ImageDataGenerator`` to apply random transformations (like rotations, flips, or color adjustments) can further diversify the dataset, making the model more resilient to variations in real-world data.

8.2.2 Model Enhancement:

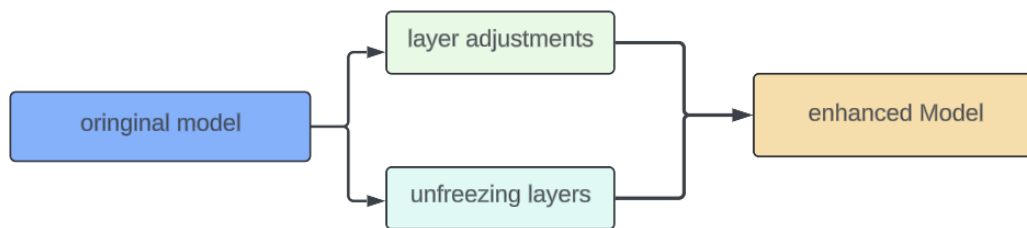


Figure 47

Layer Adjustments:

Increasing the number of layers or modifying existing ones can improve generalization. Techniques like Dropout, Batch Normalization, or adding regularization layers can help prevent overfitting.

Unfreezing Layers:

Gradually unfreezing some layers in the pre-trained model during fine-tuning can allow the model to adapt better to the new dataset while still leveraging the pre-trained knowledge, enhancing generalization.

9. Conclusion

In this project, we achieved several key milestones through a structured approach to integrating machine learning with mobile applications. Below are the main accomplishments:

1. Data Preparation: We started by processing two open-source datasets. Each image in these datasets was mapped to a corresponding label, creating a well-organized dataset for training, validation, and testing. This step was critical to ensure that the models could learn effectively from the data and that their performance could be accurately evaluated.

2. Transfer Learning: We leveraged the power of transfer learning by using MobileNetV2 as a pre-trained model. To tailor the model for our specific datasets, we added and optimized additional layers. This process involved fine-tuning parameters such as the number of epochs and the learning rate to ensure that the models achieved the required accuracy levels.

3. Model Deployment: Once trained, the models were converted into TensorFlow Lite (TFLite) format, making them compatible with mobile applications. We then developed a Flutter-based Android application that integrates these models. Users can interact with the app by uploading images from their gallery or taking pictures with their camera, which are then processed by the models to classify the images.

4. Firebase Integration: The application was further enhanced by integrating Firebase services. Firebase was used to manage user authentication, ensuring that only registered users can access the app's features. Additionally, Firebase was used to store the classification results and images, facilitating future analysis and model improvements. This setup also enables a seamless user experience, as feedback and results are consistently saved and managed through Firebase.

This project effectively combined state-of-the-art machine learning techniques with mobile application development, resulting in a functional and user-friendly application that leverages powerful cloud services.

10. Reference:

1. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520. (cite at page 8)

Available at: <https://arxiv.org/abs/1801.04381>

2. Flutter: Cross-template toolkits. (cite at page9) Available at: https://flutter.dev/multi-platform?gad_source=1&gclid=CjwKCAjw2dG1BhB4EiwA998cqE4ZlAbX595xLOfzUg6kF1gf6T38V7lrNdySvLOZvdnJd9PzMvVY_hoChUQQAvD_BwE&gclid=aw.ds

3. Firebase: Realtime database. (cite at page 10) Available at: <https://firebase.google.com/docs>

4. Google CropNet Cassava Disease Classifier. (cite at page 16) Available at: <https://www.kaggle.com/models/google/cropnet/tensorFlow2/classifier-cassava-disease-v1/1>

5. Generates a tf.data.Dataset from image files in a directory.

(cite at page 17) Available at:

https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory

6. Dataset. (cite at page 20) Available at:

<https://data.mendeley.com/datasets/tywbtsjrjv/1>

7. `tf.keras.optimizers.Adam`. (cite as page 28) Available at:

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam