

Chapter 1

Introduction

CHAPTER 1

INTRODUCTION

Agriculture has always been a basic human need ever since humans' existence as plants were a primary source of food. Even nowadays, agriculture is still considered an essential food resource and is the center of several aspects in humans' lives [1]. As a matter of fact, agriculture serves as the pillar of economy in many countries regardless of their developmental stages. The various domains that show the importance of agriculture include the fact that agriculture is a main source of livelihood where approximately 70% of the population depends on plants and their cultivation for livelihood. This great percentage reflects on agriculture being the most important resource that can actually stand a chance in the face of the rapidly increasing population [2].

One of the most critical challenges that face agriculture and affects its trade is plant diseases and how to timely detect them and deal with them to improve the health of crops. By definition, plant disease is a type of natural problems that occur in plants affecting their overall growth and might lead to plant death in extreme cases. Plant diseases can occur throughout the different stages of plant development including seed development, seedling, and seedling growth [3]. When diseased, plants go through different mechanical, morphological, and biochemical changes [4][5]. Truthfully, there are two main types of plant stress classified as biotic stress represented by living creatures that interact with plants in a way that negatively affects their growth [6] such as bacteria, viruses, or fungi [7], or abiotic stress represented by the collection of non-living factors or the environmental factors. Fig. 1 illustrates the collection of factors that contribute to plant diseases.

Typically, the commonly used approach for farmers, scientists, and even breeders, to detect and identify plant disease was the manual inspection of plants. Of course, this process requires expertise and knowledge for the proper detection. With time, manual inspection became tiresome and time consuming and not as quite efficient especially when large amounts of plants needed to be inspected. Another factor that proves the inefficiency of manual inspection is the similar conditions that might be caused by different pathogens that might look alike in their effect on the plant [6].

1.1 ABOUT PROJECT

The Crop Disease Prediction System is an AI-powered solution designed to assist agricultural professionals and farmers in identifying various crop diseases. The system leverages Deep Learning algorithms, particularly image classification models, to analyze images of crops such as leaves or stems taken in the field.

Crop diseases like leaf spot, blight, and mildew often go unnoticed until they severely impact yield and quality. Early detection and accurate classification can play a pivotal role in timely intervention and prevention of large-scale damage.

This project aims to bridge the gap between traditional visual inspection and advanced AI capabilities, providing a user-friendly platform to assist both agronomists and farmers. It combines advanced technologies such as Convolutional Neural Networks (CNNs) with pretrained model vgg16 for image-based disease detection, a web-based interface for ease of use, and a secure database for managing field and crop records.

1.2 PROJECT OBJECTIVE

The primary objectives of the Crop Disease Prediction are:

- **Automated Disease Detection:** To identify and classify common crop diseases from images using AI models.
- **Improved Accuracy:** To achieve high precision and recall rates in disease detection compared to manual methods.
- **User-Friendly Interface:** To provide a web-based platform for patients and healthcare providers to upload and analyze dental images effortlessly.
- **Educational Support:** To include resources and explanations about detected diseases for better understanding by users.
- **Patient Data Management:** To store and retrieve patient data securely for record-keeping and future references.
- **Scalability:** To design the system to handle a growing database and more complex diagnoses in the future.

Chapter 2

Software & Hardware Requirements

CHAPTER 2

SOFTWARE & HARDWARE REQUIREMENTS

2.1 SOFTWARE REQUIREMENTS

Table 2.1.1 Software Requirements

Category	Requirement	Purpose
Programming Languages	Python	Developing DL Model & Backend logic.
	JavaScript	Creating the frontend interface.
Frameworks & Libraries	TensorFlow/Keras	Building and training DL models (CNN).
	React.js	Creating the frontend interface
	Django	Backend framework for handling user requests and integrating the DL model.
	Django REST Framework	API development for connecting frontend with backend
	Tailwind CSS	Styling the web interface with responsive and modern design.
Database	Sqlite3	Secure storage and real-time management of patient data.
Development Tools	Kaggle Notebook	Cloud-based training of models with GPU/TPU support.
	Visual Studio Code	Writing and managing the frontend and backend code.

2.1 HARDWARE REQUIREMENTS

Table 2.2.1 Hardware Requirements

Usage	Requirement	Purpose
Development	Processor: Intel i5 / Ryzen 5+	For multitasking and smooth development. .
	RAM: 8 GB or more	To run IDEs and train lightweight models.
	GPU: GTX 1050 or higher	For training DL models locally.
	Storage: 256 GB SSD	Fast access and efficient file management.
Deployment (Server)	Processor: Intel Xeon or equivalent	To handle API calls and concurrent users. .
	RAM: Minimum 4 GB	For managing simultaneous interview sessions. .
	GPU (Optional): Tesla/RTX A6000	Faster model inference if needed.
	Storage: 100 GB	For storing resumes, responses, and user data securely.
End User	Device: Smartphone, PC, Tablet	End users can access via any modern device.
	Browser: Chrome, Firefox, Edge	Supports full app functionality. .
	Internet: Stable connection	Required for accessing interviews, uploading resumes, and receiving feedback.

Chapter 3

Problem Description

CHAPTER 3

PROBLEM DESCRIPTION

3.1 PROBLEM STATEMENT

In the problem statement, we have discussed some of the problems which our farmers were facing from many years. And now, we took a step to assist them with the help of technology.

From last many years, we are not getting what modern world is getting from this sector with very low area under the cultivation. We are large country by area and our 27% area is under cultivation. In last year, only 18.5% of GDP came from agriculture sector which is very low for any state whose economy is majorly based on agriculture sector.

Firstly, inefficient diagnosis of diseases because most of the farmers are illiterate and they don't not know much about the diseases. They identify the diseases on the basis of their own experience which can be inefficient and less precise so their output will certainly impact on production rate and low profit margin can damage the normal routine of farmer. Secondly, if they think about to consult agriculture expert, in majority of cases this is not possible. If they try to manage it by reaching them, then it is very time consuming and costly for the farmers to know the disease which has affected his crop. This is not convenient method at all. Thirdly, it was a big threat to food security because the country whose 25% of GDP depends on agriculture. How this nation can afford the loss of the crop? Lastly, most of the farmers of Pakistan are small-holder and if the disease effect their crop, they have no other way of earning other than this. They get all of necessary item from dealers to grow crops and in last, if their crop got ruined by disease then it's quite difficult to bear it. In current situation, 80% [9] of the agriculture crop is produced by subsistence farmers, and yield loss is around 50% [8] because of common crop diseases. These diseases can damage crops quality and wipe out entire harvests.

3.1 CHALLENGES IN THE CURRENT SYSTEM

Despite the advancements in agricultural technology, current crop disease prediction systems face several limitations and challenges:

- **Limited Access to High-Quality Data:**

Many systems lack access to diverse and well-labeled datasets, especially for less common crops and rare diseases. This limits model accuracy and generalizability across different

regions and conditions.

- **Low Accuracy in Real-World Conditions:**

Environmental factors such as lighting, background noise, and occlusion in field images can reduce the accuracy of image-based models trained under controlled settings.

- **Manual Monitoring and Delayed Detection:**

Traditional approaches still rely heavily on manual field inspections, which are time-consuming, prone to human error, and often lead to delayed disease detection.

- **Lack of Real-Time Capabilities:**

Many existing systems do not support real-time predictions, making them less effective for urgent disease outbreaks that require immediate attention

- **Inadequate User Interface and Accessibility**

Some tools are not designed with end-users (farmers) in mind, leading to low adoption due to complex interfaces or lack of local language support.

- **High Implementation Cost:**

Advanced AI-based solutions can be expensive to deploy and maintain, especially in low-resource or rural farming communities.

- **Limited Integration with Farm Management Systems**

Current solutions often operate in silos and are not integrated with broader farm management or decision support systems, reducing their overall utility

3.1 PROPOSED SOLUTION

The Crop Disease Prediction System is designed to address the challenges associated with identifying and diagnosing crop diseases by leveraging cutting-edge technology. It provides a streamlined, efficient, and accessible solution to improve agricultural diagnostics and disease management. Below is a detailed explanation of how the system addresses these challenges, aligned with the process depicted in the flowchart.

3.1.1 Automated Detection

The system employs **Convolutional Neural Networks (CNNs)** to automate the detection and classification of crop diseases. These deep learning algorithms analyze uploaded crop leaf images with precision and accuracy, identifying conditions such as spot bright, early bright,

scrab etc. This automation minimizes the potential for human error, ensuring more consistent and reliable crops for farmers.

Flowchart Alignment: After the user uploads an image, the "Plantify-AI Processes the Image" step leverages CNN models to extract and interpret features from the image for disease classification.

3.1.2 User-Friendly Platform

The system offers an **intuitive interface** that simplifies the process for users. Patients can easily navigate the platform to upload their dental images and receive real-time crop results. The system ensures seamless user experience, reducing the technical barriers that might prevent farmers from accessing crop [5].

Flowchart Alignment: The step "User Upload Image" demonstrates the ease with which users can interact with the platform. The subsequent steps, such as displaying predictions and accuracy, ensure that results are delivered in an understandable format.

3.1.3 Global Accessibility

Hosting the system on a **web-based platform** ensures global scalability and accessibility. This approach is particularly beneficial in underserved or remote regions where access to dental professionals may be limited. By providing an online solution, the system enables a larger audience to access essential tools, thus bridging agriculture gaps [2].

Flowchart Alignment: The entire workflow—from image upload to displaying results—is designed for online deployment, ensuring that users from any part of the world can utilize the service with minimal infrastructure requirements.

3.1.4 AI-Powered Support

By integrating AI-powered tools, the system reduces dependency on manual inspection and agricultural expertise. Farmers and agronomists can use this tool as an assistant, enabling more accurate and consistent identification of crop diseases. The AI's capability to process and analyze vast amounts of image data ensures a higher level of diagnostic confidence, complementing traditional agricultural practices.

Flowchart Alignment: The "Model Predicts Result" step highlights the AI's role in generating accurate predictions based on the processed crop images, assisting farmers and agricultural experts in identifying diseases and making informed decisions for timely intervention.

3.1.5 Cost-Effective Solution

The system is designed to be a practical and affordable solution for both farmers and agricultural experts. By leveraging advanced technology, it reduces the costs associated with traditional disease detection methods while maintaining high accuracy and reliability. This affordability encourages widespread adoption, making plant health management more accessible and inclusive across diverse farming communities.

Flowchart Alignment: The platform's simple workflow—from capturing and uploading crop images to receiving disease predictions—minimizes operational complexity, keeping costs low without compromising accuracy and effectiveness.

3.2 WORKFLOW OF THE CROP DISEASE PREDICTION SYSTEM

Start

The process begins when a user interacts with the Crop Disease Prediction System. This marks the initiation of the automated prediction workflow for Crop images.

User Uploads Image

The first interaction involves the user uploading an image, such as a crop leaf images. This image serves as the input for the AI system to analyze. The user-friendly interface ensures the process is seamless and accessible.

Plentify AI Process the Image

The uploaded image is processed by the AI system, referred to as "Plentify AI." This step involves various preprocessing techniques like resizing, contrast enhancement, and normalization. These processes prepare the image for accurate analysis by the deep learning model.

Model Makes Prediction

Using pretrained model vgg16, the AI model processes the image and predicts the outcome. The model classifies the crop condition (e.g., early bright, healthy, light spot, etc.) based on its training on annotated datasets.

Display Prediction & Confidence Score

The system displays prediction results to the user, accompanied by the model's confidence level. This transparency helps users understand the reliability of the diagnosis and serves as a decision-making aid for both patients and professionals.

End

The workflow concludes once the results are presented. The user can take further action, such as consulting a dentist or exploring treatment options, based on the diagnostic insights provided by the system.

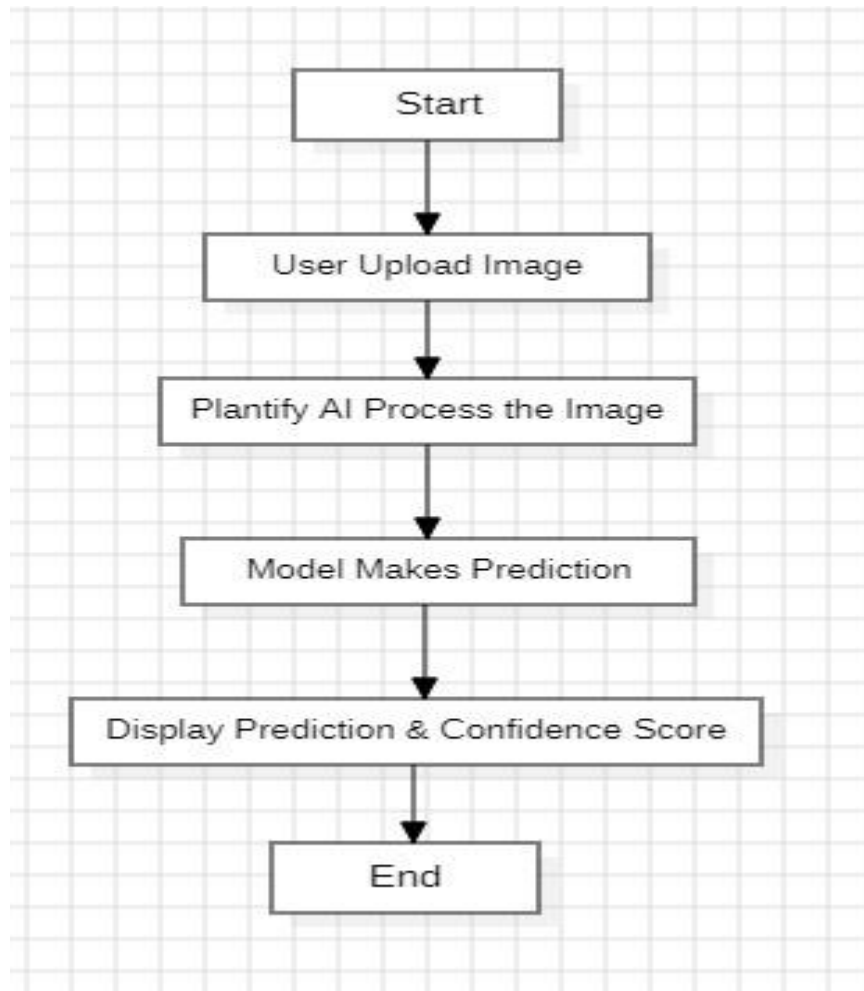


Figure 3.3.1 Workflow of Project

Chapter 4

Literature Survey

CHAPTER 4

LITERATURE SURVEY

LITERATURE SURVEY

Ferentinos [11] used an open database of 87,848 leaf images for performing a set of experiments. The images were captured from healthy as well as diseased plants. The dataset includes 38 different classes and 14 species of peach, pepper, apple, raspberry, soybean, etc. Its training dataset contains 70,300, and the testing dataset contains 17,548 images. The author applied five different models to this dataset. They achieved an accuracy of 99.06% by using AlexNet, 92.27% by using GoogLeNet, and 99.53% accuracy by applying the VGG model. The authors performed a set of experiments on original images and pre-processed images. All models yielded higher accuracy on original images than pre-processed images. However, a significant reduction in computation time is noticed on applying the above-stated models on pre-processed, down-scaled, and squared images [11].

VGG16 [11] is devised by the Visual Geometry Group from the University of Oxford. It was the first runner-up in the ILSVRC-2014 challenge. The localization and classification accuracy of this model increases with an increase in the depth of a model. VGG16 is a simpler network. It uses small convolutional filters of dimensions 3×3 with a stride of one in all layers. It includes a max-pooling layer of dimension 2×2 with a stride of two. It receives an RGB image of dimension $224 \times 224 \times 3$ as an input. In the training dataset, the mean average value of RGB is subtracted from each pixel to perform pre-processing. A pre-processed image is passed through a stack of convolutional layers followed by five max-pooling layers. It uses the first two fully connected layers with 4096 channels in each layer and the third fully connected layer with 1000 channels. The last layer performs a 1000-way classification. At the last stage, a softmax layer works to determine multi-class probabilities.

Andreas Kamilaris et. al.[2018], author discussed and perform a survey of 40 researches efforts that employ deep learning techniques, applied to various agricultural and food production challenges. To study the agricultural problems stated under each work, the specific models and frameworks employed the sources, nature and used dataset and overall performance achieved with help of used methods, comparison of deep learning with other techniques also done. It is stated that deep learning provides high accuracy by using image processing techniques [10].

Konstantinos P. Ferentinos et. al.[2018], author discussed about convolutional neural network models were developed to perform plant disease detection and diagnosis through deep learning methodologies using plants leaves images. Training of proposed models was done with the use of an open database of 87,848 images, containing 25 different plants in a set of 58 distinct classes of [plant, disease] combinations. Some models get success 92% in detecting the corresponding combination [plant, disease]. High rate of success model is very useful to early detection tool and have possibilities of further expanded by researchers [11].

Basvaraj S. Anami et al in 2011, proposed better machine vision system in the area of disease recognition, both the feature color and texture are used to recognize and classify different agriculture product into normal and affected using neural network classifier [12].

Theobroma cocoa is a plant native to the tropical regions of Central and South America. It thrives in equatorial climates and is cultivated in tropical regions across West and Central Africa, Latin America, and Southeast Asia [1]. *Theobroma cocoa* produces cocoa pods which have an oval-shaped fruit of which the bark is about 80% and the rest, 20%, is made up of seeds known as cocoa beans and white pulp known as mucilage [2]. Cocoa beans are processed into chocolate, cocoa powder, cocoa drinks, and other related products in the food industry, as well as products for the pharmaceutical and cosmetic industries [3].

Agricultural biodiversity is foundational to providing food and raw materials to humans. When pathogenic organisms such as fungi, bacteria, and nematodes; the soil pH; temperature extremes; changes in the amount of moisture and humidity in the air; and other factors continuously disrupt a plant, it can develop a disease. Various plant diseases can impact the growth, function, and structures of plants and crops, which automatically affect the people who are dependent on them. The majority of farmers still use manual methods to identify plant illnesses, since it is challenging to do so early on and has a negative impact on productivity. To overcome this, many deep learning (DL), image processing, and machine learning (ML) techniques are being developed, by which the detection of disease in a plant is performed by images of plant leaves.

Chapter 5

Software Requirement and Specification

CHAPTER 5

SOFTWARE REQUIREMENTS SPECIFICATION

5.1 FUNCTIONAL REQUIREMENTS

Functional requirements specify the key operations and behaviors the system must perform to achieve its goals.

Table 5.1.1 Functional Requirements

Sr. No.	Requirement Name	Description
1.0	User Authentication and Account Management	<ol style="list-style-type: none"> 1. User navigates to the login or registration page. 2. User creates an account by providing basic details like email and password. 3. User logs in securely using the provided credentials. 4. Users can update profile details, reset passwords, and manage account settings. 5. Verification process (e.g., OTP) is included during sign-up to ensure authenticity.
2.0	Dashboard Access	<ol style="list-style-type: none"> 1. After login, the user is redirected to a personalized dashboard. 2. The dashboard provides options to upload images, view diagnostic results, view profile and logout.
3.0	Image Upload	<ol style="list-style-type: none"> 1. User navigates to the dashboard. 2. User clicks on upload or drags and drops an image file. 3. System provides options for cropping and adjusting the image. 4. User saves the cropped image, and it is used for processing and reference.
4.0	Image Preprocessing	<ol style="list-style-type: none"> 1. Uploaded images are resized to a standard resolution. 2. Filters are applied to enhance image quality. 3. Images are normalized for consistent analysis.
5.0	Disease Prediction and Classification	<ol style="list-style-type: none"> 1. User submits an image for disease analysis. 2. System processes the image using a deep learning model. 3. System predicts the disease (e.g., cavities, implants) with a confidence score.
6.0	Disease Results	<ol style="list-style-type: none"> 1. User selects the processed image on the

7.0	Display	<p>dashboard.</p> <ol style="list-style-type: none"> 1. System displays results in a user-friendly format. 2. Users are alerted when results are ready. 3. Visualization options are provided for better understanding.
8.0	Contact and Support	<ol style="list-style-type: none"> 1. User accesses the support option from the dashboard. 2. User submits a query or message to the admin. 3. Admin responds to user queries and messages.
9.0	Database Management	<ol style="list-style-type: none"> 1. The system securely stores user data. 2. The system maintains efficient records for easy retrieval and management.

5.2 NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements define the overall system attributes such as performance, usability, and reliability.

System Availability

The system is highly reliable and accessible, with a minimum up time of 99.5%. This ensures users can access the system with minimal downtime, critical for continuous operations in diagnostic tasks.

Response Time

The system processes user input and delivers Crop disease results within 5 seconds. This guarantees fast and efficient user experience, reducing waiting times and ensuring smooth functionality.

Concurrent User Support

The system supports 50 concurrent users without noticeable performance degradation. It manages multiple users simultaneously, ensuring scalability for larger user bases.

Responsive User Interface

The system provides a responsive design that adapts to different devices, including mobile phones, tablets, and desktops. This ensures users can access the platform seamlessly across various screen sizes and devices.

Data Security

The system follows security standards, such as hashing login credentials to protect privacy. Secure communication protocols like HTTPS implemented to prevent unauthorized access or data breaches.

Compliance with Data Privacy Regulations

The system adheres to relevant data protection laws, including GDPR and HIPAA, to ensure lawful handling of personal user data. This includes measures like obtaining user consent, ensuring data anonymity, and providing options for data deletion.

Scalability

The system is scalable to handle an increasing number of users, larger datasets, or more complex processes without performance degradation. It allows for smooth scaling in terms of both infrastructure and application capabilities

Documentation

The system includes comprehensive and detailed documentation for end users and developers. This should cover system functionality, API usage, troubleshooting guides, and user instructions to ensure smooth operation and maintenance.

Chapter 6

Software Design

CHAPTER 6

SOFTWARE DESIGN

6.1 USECASE DIAGRAM

The Use Case Diagram illustrates the various interactions between users (actors) and the system. It identifies different use cases and describes how users interact with the system to achieve specific goals. In the context of the Crop Disease Prediction System, the Use Case Diagram may include actors such as the user imaging device or camera sensor, image processing module, and output/visualization component.

Actors

User: This actor interacts with the system for tasks like signing up, logging in, viewing profiles, predicting results, and contacting support.

Server: The server has processed the image and predicted the result.

Use Cases for User

Signup: Users can sign up to the system.

Login: Users log in to access the system.

View Profile: Users can view their profile information.

Dashboard: Users access the dashboard after logging in.

Predict: Users upload images to predict crop diseases.

Result: Users view the results of their predictions.

Contact: Users can contact support or teams.

Use Cases for server

Image processing: server has processed the image.

Predict result: server has predicted the crop disease.

Include and Extend Relationships

Include: The "Verification" process, which includes OTP, is required when signing up. It extends the **Profile** use case, suggesting that profile details are modified or edited after verification.

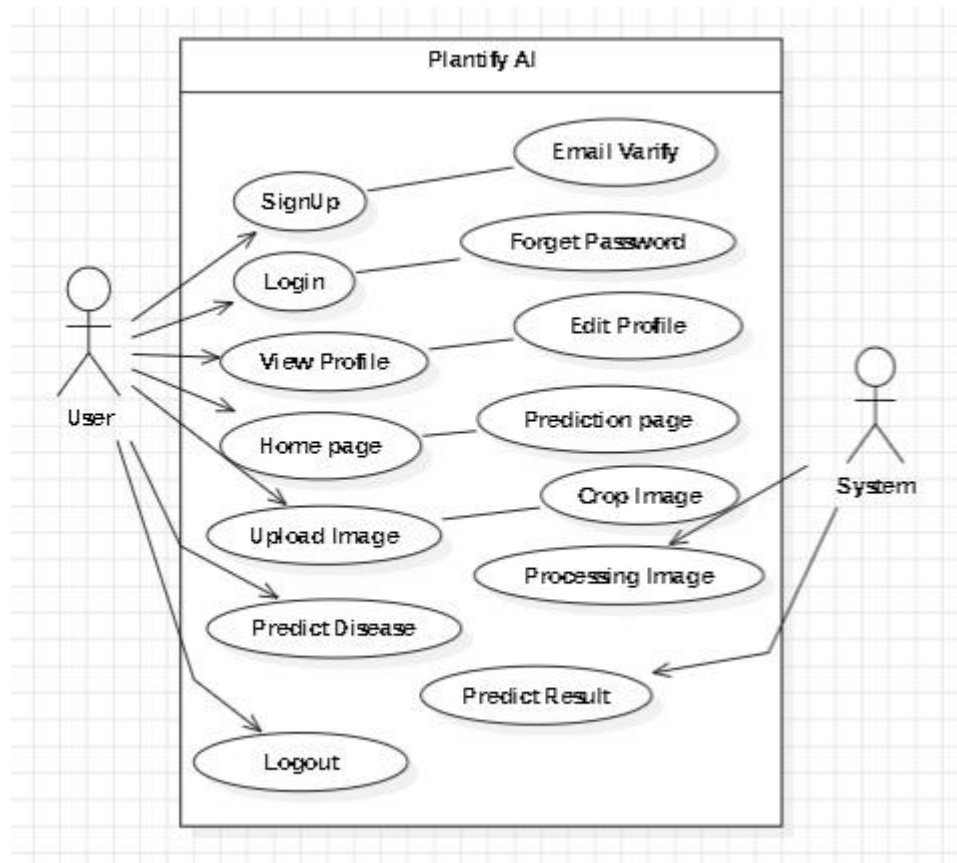


Figure 6.1.1 : Use Case Diagram

6.2 DATA FLOW DIAGRAM (DFD)

The Data Flow Diagram (DFD) provides a visual representation of how data flows through the Crop Disease Prediction System. It illustrates the processes, data stores, and data flows within the system, highlighting the transformation of data at each stage. In the context of the Crop Disease Prediction System, the DFD may depict the flow of data from the imaging device or camera sensor to the preprocessing module, crop disease prediction process, feature extraction, and output visualization or report generation.

Classes and Interactions

Plantify-AI: It represents the system where the user uploads an image for disease prediction. It interacts with both the model and the user. It predicts the image and shows the accuracy.

User: Users interact with the system by uploading images for prediction. The class handles image upload and interacts with plantify-AI to get predictions.

Image Preprocessing Unit: This Entity preprocesses the real time image and converts it to model acceptable format by using various preprocessing steps such as resizing, contrast enrichment, normalization etc.

Model: This class represents the Deep Learning Model used for crop disease prediction via image data. It outputs the predicted disease name and confidence score.

Interactions

The **User** uploads an image, which is sent to **Plentify AI**.

Plentify AI processes the image using the image preprocessing unit, the preprocessed image will be given to the Deep Learning **Model**, which then outputs a disease name and confidence score.

The **Plentify AI** shows the prediction and the score to the user.

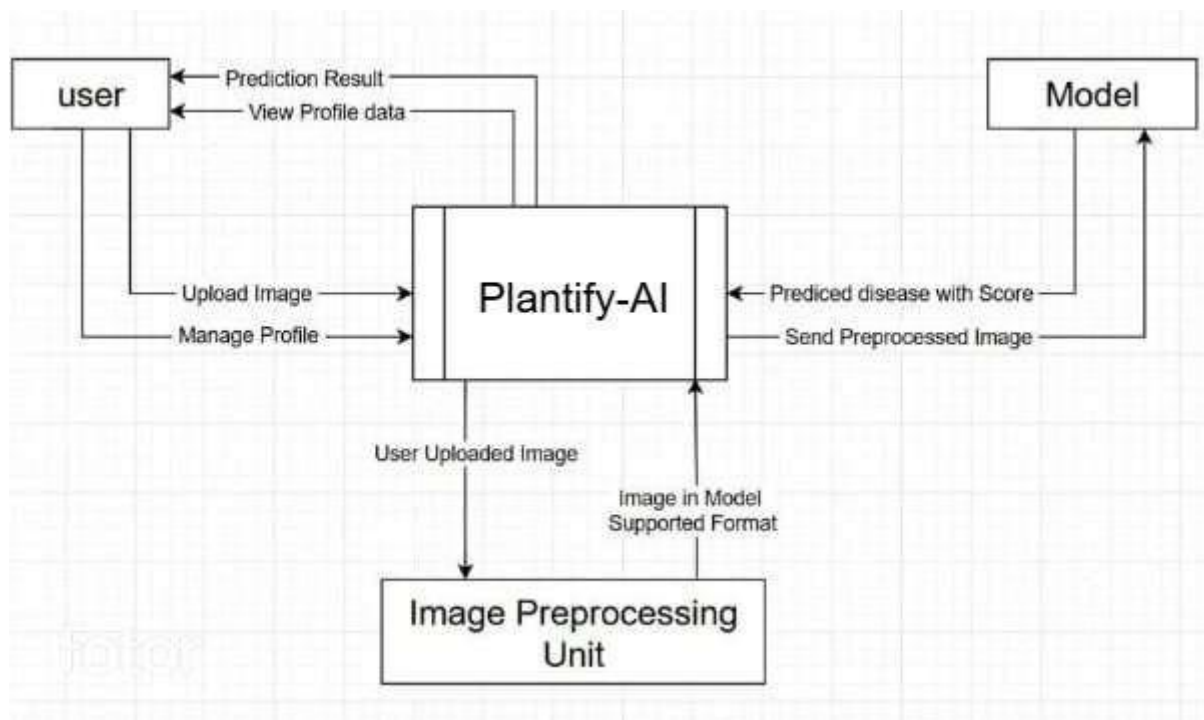


Figure 6.2.1 : Data Flow Diagram level 0

CHAPTER-7

DEEP

LEARNING MODULE

CHAPTER 7

DEEP LEARNING MODULE

7.1 DATASET DESCRIPTION

This dataset is derived from New Plant Disease Dataset, which offers a comprehensive collection of high-resolution images of various plant species. In this enhanced version, each image has been meticulously preprocessed to isolate individual leaves or affected regions, facilitating focused analysis of plant health conditions such as fungal infections, nutrient deficiencies, pest damage, and viral diseases.

This dataset is organized into two primary directories: train, and valid. Each of these directories contains subfolders representing specific plant conditions, including Leaf Blight, Rust, Powdery Mildew, Bacterial Spot, Healthy. Each image within the dataset is labelled based on the visible condition of the plant leaf, enabling precise classification and analysis.

The original dataset consisted of dental radiographs grouped into the following classes:

Table 7.1.1: Different types of crops and data instances.

S.No	Plant Disease	Sample Instances
1.	Apple Apple Scab	2015
2.	Apple Black Rot	1986
3.	Apple Cedar Apple Rust	1762
4.	Apple Healthy	2005
5.	Blueberry Healthy	1814
6.	Cherry Healthy	1726
7.	Cherry Powdery Mildew	1683
8.	Corn Cercospora Leaf Spot	1642
9.	Corn Common Rust	1907
10.	Corn Healthy	1859
11.	Corn Northern Leaf Blight	1908

12.	Grape Black Rot	1888
13.	Grape Esca	1920
14.	Grape Healthy	1692
15.	Grape Leaf Blight	1722
7.2 7.3 16.	Orange Huanglongbing	2010
7.4 17.	Peach Bacterial Spot	1838
18.	Peach Healthy	1728
19.	Pepper Bacterial Spot	1913
20.	Pepper Healthy	1988
21.	Potato Early Blight	1939
22.	Potato Healthy	1824
23.	Potato Late Blight	1939
24.	Raspberry Healthy	1781
25.	Soybean Healthy	2022
26.	Squash Powdery Mildew	1736
27.	Strawberry Healthy	1824
28.	Strawberry Leaf Scorch	1774
29.	Tomato Bacterial Spot	1702
30.	Tomato Early Blight	1920
31.	Tomato Healthy	1926
32.	Tomato Late Blight	1851
33.	Tomato Leaf Mold	1882
34.	Tomato Septoria Leaf Spot	1745
35.	Tomato Spider mites	1741
36.	Tomato Target Spot	1827
37.	Tomato mosaic	1790
38.	Tomato Yellow Leaf Curl	1961

7.2 PREPROCESSING STEP

7.2.1 Resize image

Resize the images of 224 x 224 x 3 size.

7.2.2 Normalization

Scaling pixel values to a range of 0-1 to improve model performance.












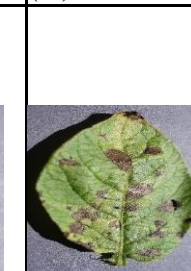
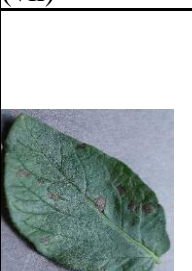





7.3 DATA VISULIZATION

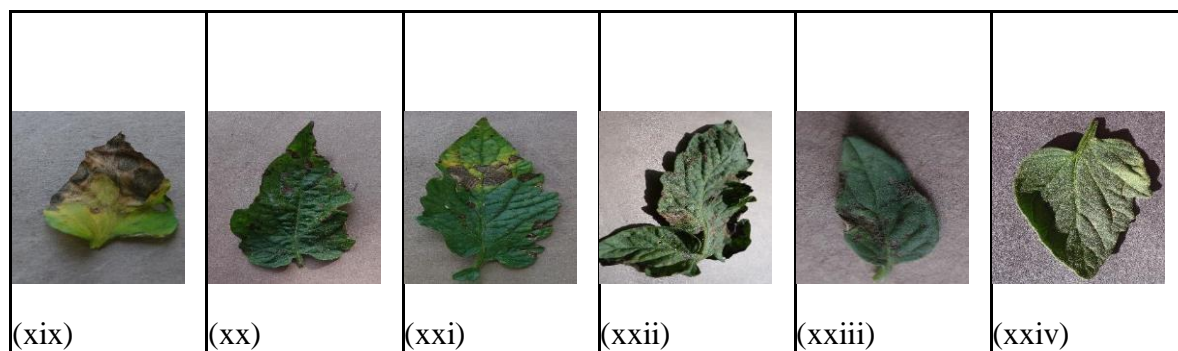
Data visualization helps in understanding the distribution and characteristics of the dataset. Key visualizations include.

- **Sample Images:** Displaying a few examples from each class to understand image variability.

Table 7.3.2: Description of train dataset.

7.4

					
(i)	(ii)	(iii)	(iv)	(v)	(vi)
					
(vii)	(viii)	(ix)	(x)	(xi)	(xii)
					
(xiii)	(xiv)	(xv)	(xvi)	(xvii)	(xviii)



7.4 DL MODEL DESCRIPTION

The VGG-16 model is a convolutional neural network (CNN) architecture that was proposed by the Visual Geometry Group (VGG) at the University of Oxford. It is characterized by its depth, consisting of 16 layers, including 13 convolutional layers and 3 fully connected layers. VGG-16 is renowned for its simplicity and effectiveness, as well as its ability to achieve strong performance on various computer vision tasks, including image classification and object recognition. The model's architecture features a stack of convolutional layers followed by max-pooling layers, with progressively increasing depth. This design enables the model to learn intricate hierarchical representations of visual features, leading to robust and accurate predictions. Despite its simplicity compared to more recent architectures, VGG-16 remains a popular choice for many deep learning applications due to its versatility and excellent performance.

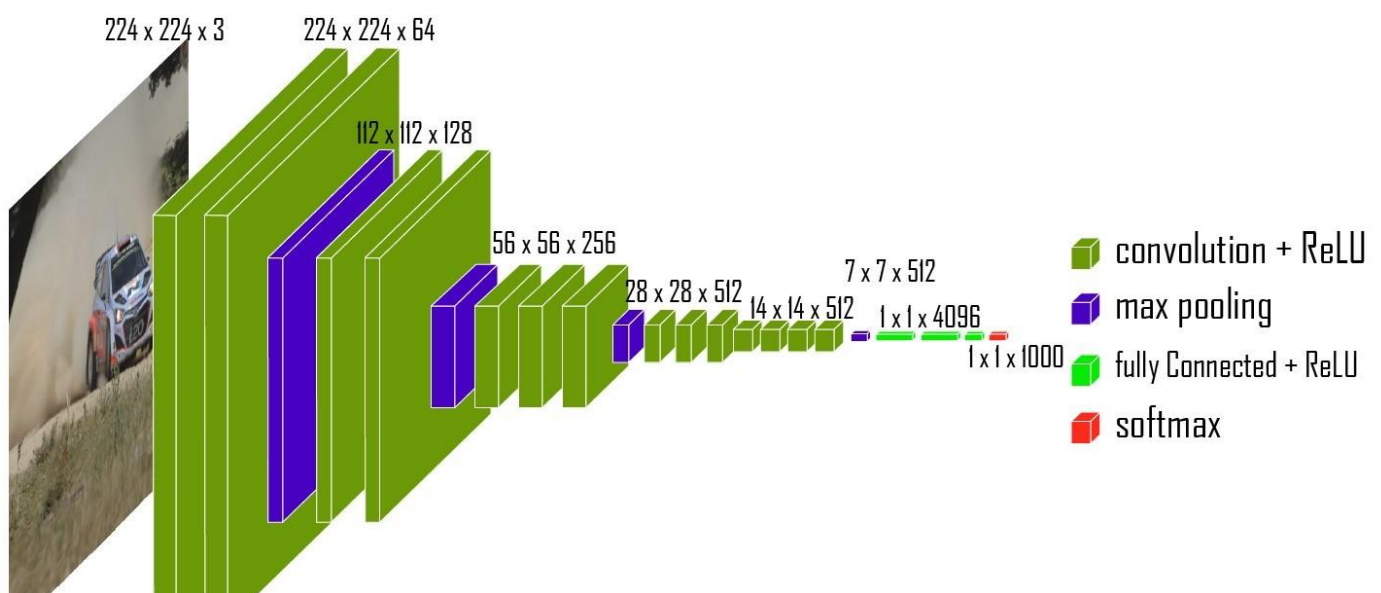


Figure 7.4.1 VGG16 Architecture

1. Input Layer:

- Input dimensions: (224, 224, 3)

2. Convolutional Layers (64 filters, 3×3 filters, same padding):

- Two consecutive convolutional layers with 64 filters each and a filter size of 3×3.
- Same padding is applied to maintain spatial dimensions.

3. Max Pooling Layer (2×2, stride 2):

- Max-pooling layer with a pool size of 2×2 and a stride of 2.

4. Convolutional Layers (128 filters, 3×3 filters, same padding):

- Two consecutive convolutional layers with 128 filters each and a filter size of 3×3.

5. Max Pooling Layer (2×2, stride 2):

- Max-pooling layer with a pool size of 2×2 and a stride of 2.

6. Convolutional Layers (256 filters, 3×3 filters, same padding):

- Two consecutive convolutional layers with 256 filters each and a filter size of 3×3.

7. Convolutional Layers (512 filters, 3×3 filters, same padding):

- Two sets of three consecutive convolutional layers with 512 filters each and a filter size of 3×3.

8. Max Pooling Layer (2×2, stride 2):

- Max-pooling layer with a pool size of 2×2 and a stride of 2.

9. Stack of Convolutional Layers and Max Pooling:

- Two additional convolutional layers after the previous stack.
- Filter size: 3×3.

10. Flattening:

- Flatten the output feature map (7×7×512) into a vector of size 25088.

11. Fully Connected Layers:

- Three fully connected layers with ReLU activation.
- First layer with input size 25088 and output size 4096.
- Second layer with input size 4096 and output size 4096.
- Third layer with input size 4096 and output size 1000, corresponding to the 1000

classes in the ILSVRC challenge.

- Softmax activation is applied to the output of the third fully connected layer for classification.

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 512)	0
dense_6 (Dense)	(None, 512)	262,656
batch_normalization_3 (BatchNormalization)	(None, 512)	2,048
dropout_3 (Dropout)	(None, 512)	0
dense_7 (Dense)	(None, 38)	19,494

Total params: 14,998,886 (57.22 MB)

Trainable params: 283,174 (1.08 MB)

Non-trainable params: 14,715,712 (56.14 MB)

Figure 7.4.2 Model Summary

- **Model Architecture:** CNN architecture for feature extraction, train for crop disease prediction.
- **Input:** Preprocessed images of size 224 x 224 x 3.
- **Output:** Class probabilities for each disease type
- **Training Details:**
 - Loss Function: Categorical Cross Entropy.
 - Optimizer: Adam optimizer with a learning rate of 0.001
 - Epochs: 10.
 - Metrics: Accuracy, Precision, Recall, F1-Score.

7.5 RESULT ANALYSIS

The performance of the model is evaluated using various metrics

1. **Accuracy:** Measures the proportion of correct predictions out of total predictions.
 - Training Accuracy: 94%.

- Validation Accuracy: 92%

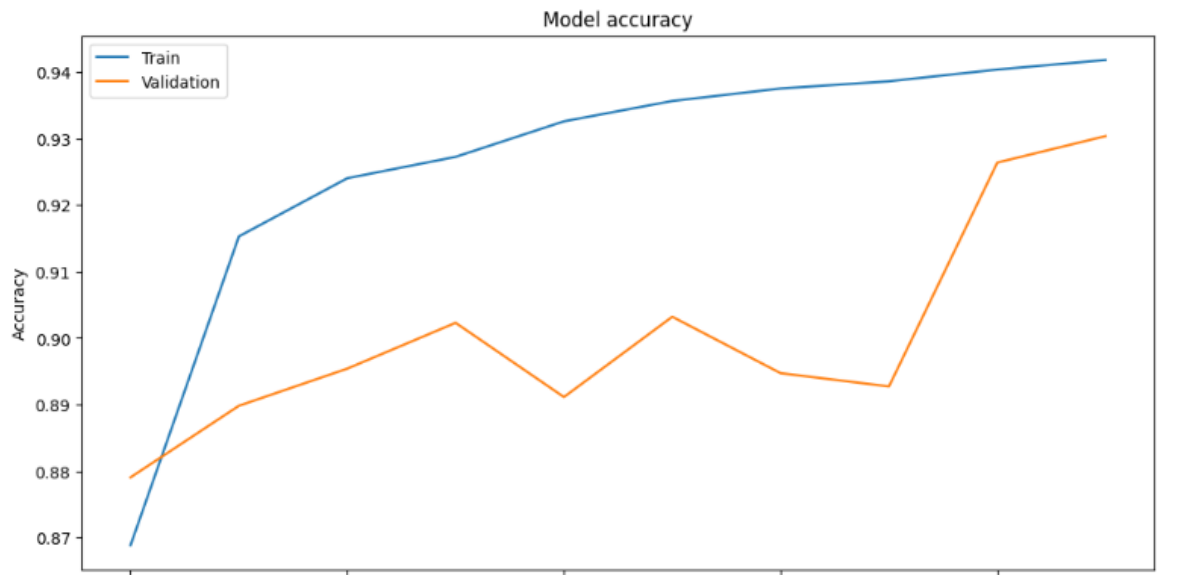


Figure 7.5.2 Model Accuracy

- 2. Loss Graphs:** Training and validation loss over epochs to assess overfitting or underfitting.

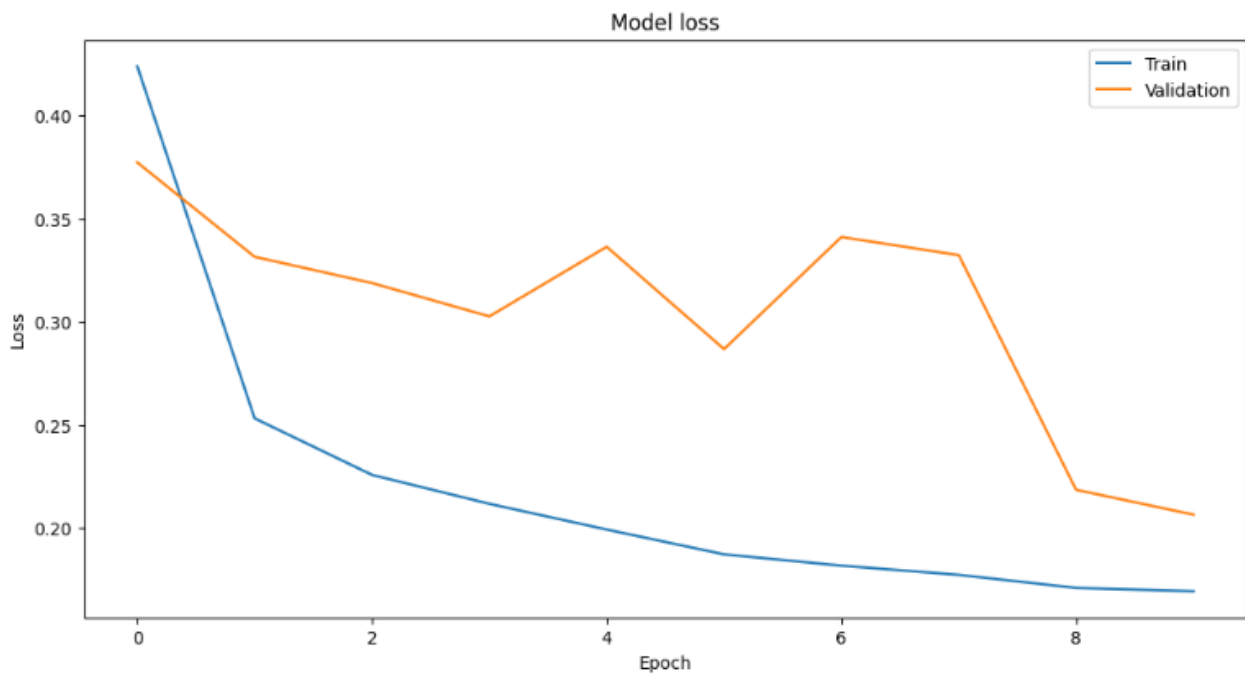


Figure 7.5.3 Training vs Validation Loss

Chapter 8

Front End connectivity

CHAPTER 8

FRONTEND CONNECTIVITY

This chapter describes how the front-end of the Teeth Disease Classification System connects and interacts with the backend and deep learning modules to provide seamless user experience.

8.1 FRONT-END TECHNOLOGY STACK

The front-end is designed to be user-friendly and responsive, enabling users to interact efficiently with the system. The technologies used are:

- **React.js:** For building dynamic and reusable user interface components.
- **Vite:** As the frontend build tool for fast development and bundling.
- **Tailwind CSS:** For responsive, utility-first CSS styling.
- **JavaScript (ES6+):** To handle dynamic behaviour and interactivity.
- **Axios:** For making HTTP requests and handling API communication.

8.2 FRONT-END FEATURES

The system provides the following key features on the client side:

1. User Registration and Login:

- Users can register with email or phone.
- Password-based secure login mechanism.
- Session handled using JWT tokens Dashboard:

2. Disease Prediction:

- An upload interface where users can submit images for analysis.

3. Result Display:

- A visually appealing page that presents prediction results, confidence scores, and recommendations.

4. Profile Management

Users can update:

- First Name, Last Name
- Phone Number
- Date of Birth
- Profile image

-

8.3 BACKEND INTEGRATION

The frontend connects to the backend using RESTful APIs served by Django REST Framework. Interaction flows are as follows:

1. Image upload

- Users upload image via the frontend.
- Image is forwarded to backend APIs for processing and evaluation

2. Authentication and Profile

- Token-based authentication ensures secure user sessions.
- APIs are used to fetch and update user profiles dynamically.

8.5 TESTING & DEBUGGING

Frontend Testing

- API routes tested using Postman and Swagger UI
- Ensures all endpoints return expected results with proper error handling.

Debugging Tools

- Utilized Chrome DevTools for real-time inspection of:
 - Network requests
 - Component rendering
 - Console logs and warnings

Error Handling

- Implemented UI-level alerts for:
 - Invalid credentials
 - Invalid credentials
 - Server errors

DEEP LEARNING MODULE INTEGRATION

The front-end interacts with the DL module through the backend:

Image Upload:

- Users upload images via the front-end.
- The image is sent to the backend, where it undergoes preprocessing before being passed to the DL model.

5. **Result Retrieval:**

- The prediction output from the DL model (class label and confidence score) is returned to the front-end via the backend and displayed to the us

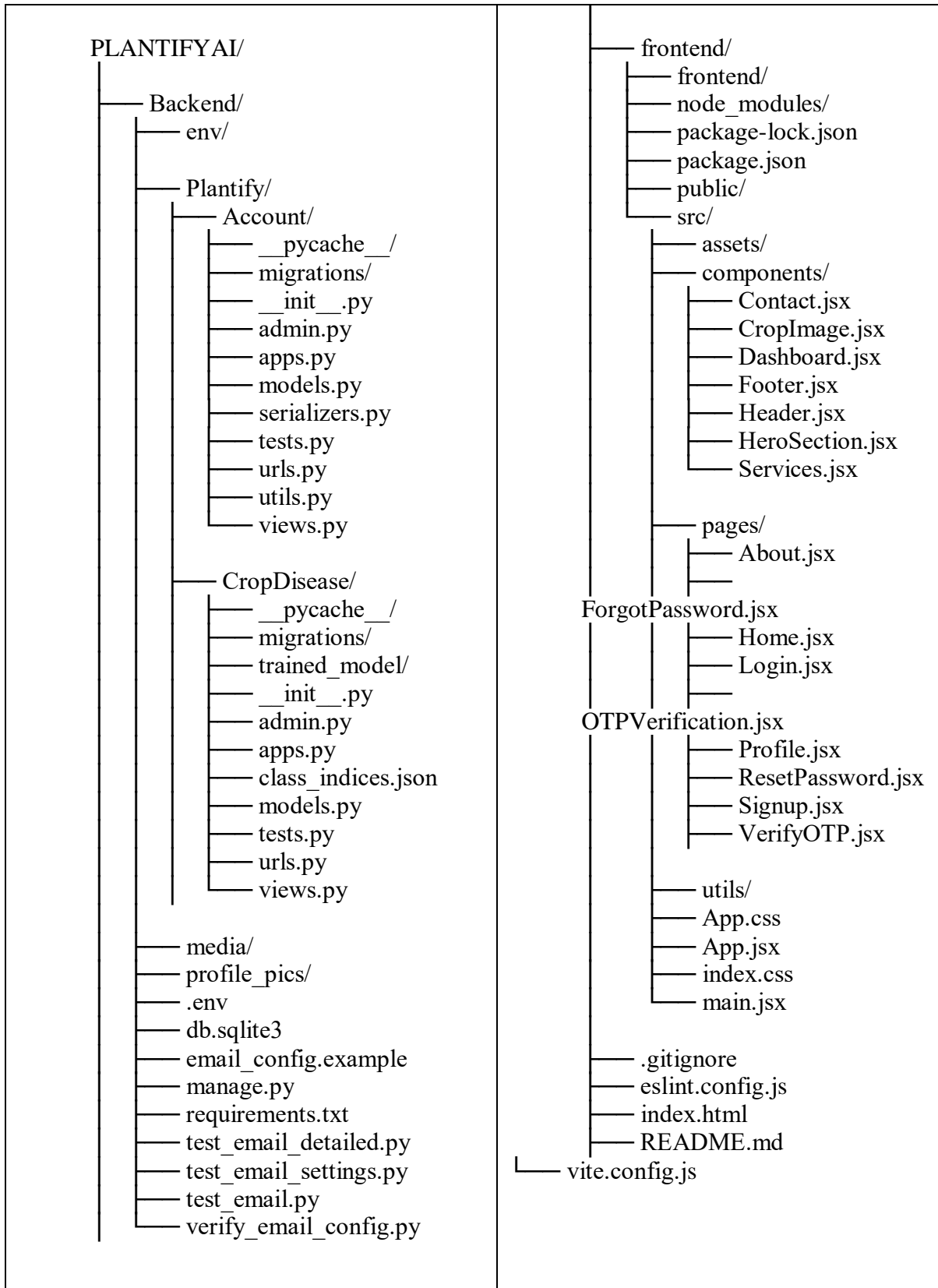
CHAPTER-9

CODING

CHAPTER 9

CODING

9.1 STRUCTURE OF PROJECT & APP



9.2 CODING OF PROJECT DENTIFY AI

Account/views.py

```

from django.shortcuts import render, redirect
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import authentication, permissions
from django.contrib.auth import get_user_model, authenticate, login
from rest_framework import status
from .serializers import RegisterSerializer, EmailVerificationSerializer, ResendOTPSerializer,
PasswordResetRequestSerializer, PasswordResetVerifySerializer, PasswordResetConfirmSerializer
from rest_framework.authtoken.models import Token
from django.core.validators import validate_email
from django.core.exceptions import ValidationError
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer
from django.contrib.auth.decorators import login_required
from .models import UserProfile, EmailVerification, PasswordResetToken
from rest_framework.permissions import IsAuthenticated
from django.contrib.auth.models import User # Import the built-in User model
from .serializers import ProfileSerializer
from django.utils import timezone
from django.core.mail import send_mail
from django.conf import settings
from .utils import send_otp_email, validate_password_strength, get_password_strength_score,
send_email_with_fallback

User = get_user_model()

class MyTokenObtainPairSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)
        token['email'] = user.email
        token['first_name'] = user.first_name
        token['last_name'] = user.last_name
        return token

class MyTokenObtainPairView(TokenObtainPairView):
    serializer_class = MyTokenObtainPairSerializer

class MyTokenRefreshView(TokenRefreshView):
    pass

class RegisterView(APIView):
    def post(self, request):
        try:
            data = request.data

            # Validate required fields
            required_fields = ['email', 'password', 'password2', 'first_name', 'last_name']
            for field in required_fields:
                if not data.get(field):
                    return Response(
                        {'error': f'{field} is required'},
                        status=status.HTTP_400_BAD_REQUEST
                    )

            # Validate email format

```

```

try:
    validate_email(data['email'])
except ValidationError:
    return Response(
        {'error': 'Invalid email format'},
        status=status.HTTP_400_BAD_REQUEST
    )

# Check if email exists
if User.objects.filter(email=data['email']).exists():
    return Response(
        {'error': 'Email already exists'},
        status=status.HTTP_400_BAD_REQUEST
    )

# Validate password match
if data['password'] != data['password2']:
    return Response(
        {'error': 'Passwords do not match'},
        status=status.HTTP_400_BAD_REQUEST
    )

# Validate password strength
is_valid, errors = validate_password_strength(data['password'])
if not is_valid:
    return Response(
        {'error': 'Password is not strong enough', 'details': errors},
        status=status.HTTP_400_BAD_REQUEST
    )

# Create user
user = User.objects.create_user(
    email=data['email'],
    username=data['email'], # Set username same as email
    password=data['password'],
    first_name=data['first_name'],
    last_name=data['last_name'],
    is_active=True, # Set to False if you want to prevent login until email is verified
)

# Create user profile
UserProfile.objects.create(
    user=user,
    bio="",
    address="",
    phone=""
)

# Generate OTP and create verification record
otp = EmailVerification.generate_otp()
verification = EmailVerification.objects.create(
    user=user,
    otp=otp,
    expires_at=timezone.now() + timezone.timedelta(minutes=10)
)

# Send OTP to user's email
email_sent = send_otp_email(user.email, otp)

# Generate JWT tokens
refresh = MyTokenObtainPairSerializer.get_token(user)

response_data = {
    'success': 'User registered successfully',
    'data': {

```

```

        'user_id': user.id,
        'email': user.email,
        'first_name': user.first_name,
        'last_name': user.last_name,
        'requires_verification': True,
        'email_sent': email_sent
    }
}

# Include OTP in response during development
if settings.DEBUG:
    response_data['data']['development_otp'] = otp
    response_data['data']['note'] = 'This OTP is only included in DEBUG mode'

return Response(response_data, status=status.HTTP_201_CREATED)

except Exception as e:
    return Response(
        {'error': str(e)},
        status=status.HTTP_400_BAD_REQUEST
    )

def send_verification_email(self, email, otp):
    subject = 'Verify your PlantifyAI account'
    message = f'Your verification code is: {otp}\n\nThis code is valid for 10 minutes.'
    from_email = settings.EMAIL_HOST_USER
    recipient_list = [email]

    # Print OTP clearly in console for development
    print("\n" + "="*50)
    print(f"✉ VERIFICATION EMAIL FOR: {email}")
    print(f"🔑 OTP CODE: {otp}")
    print("="*50 + "\n")

    try:
        send_mail(subject, message, from_email, recipient_list)
        return True
    except Exception as e:
        print(f"Error sending email: {e}")
        return False

class LoginView(APIView):
    def post(self, request):
        try:
            data = request.data
            email = data.get('email')
            password = data.get('password')

            if not email or not password:
                return Response(
                    {'error': 'Please provide both email and password'},
                    status=status.HTTP_400_BAD_REQUEST
                )

            # Get user by email
            try:
                user = User.objects.get(email=email)
            except User.DoesNotExist:
                return Response(
                    {'error': 'User not found'},
                    status=status.HTTP_404_NOT_FOUND
                )

            # Check password
            if not user.check_password(password):

```



```

        return Response(
            return Response({
                'error': 'Invalid OTP',
            }, status=status.HTTP_400_BAD_REQUEST)

    # Check if OTP is expired
    if not verification.is_valid():
        return Response({
            'error': 'OTP expired',
        }, status=status.HTTP_400_BAD_REQUEST)

    # Mark as verified
    verification.is_verified = True
    verification.save()

    # Update user verification status
    profile.is_email_verified = True
    profile.save()

    # Generate tokens for automatic login
    refresh = MyTokenObtainPairSerializer.get_token(user)

    return Response({
        'success': 'Email verified successfully',
        'data': {
            'user_id': user.id,
            'email': user.email,
            'access': str(refresh.access_token),
            'refresh': str(refresh)
        }
    }, status=status.HTTP_200_OK)

except User.DoesNotExist:
    return Response({
        'error': 'User not found',
    }, status=status.HTTP_404_NOT_FOUND)
except Exception as e:
    return Response({
        'error': str(e),
    }, status=status.HTTP_400_BAD_REQUEST)

```

Ac

count/urls.py

```

from django.urls import path
from .views import *
from Account import views
from .views import UserProfileView, VerifyOTPView, ResendOTPView, CheckPasswordStrengthView,
PasswordResetRequestView, PasswordResetVerifyView, PasswordResetConfirmView

from rest_framework_simplejwt.views import (
    TokenRefreshView,
)

urlpatterns = [
    path('token/', views.MyTokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('register/', views.RegisterView.as_view(), name='auth_register'),
    path('login/', views.LoginView.as_view(), name='auth_login'),
    path('profile/', views.UserProfileView.as_view(), name='profile'),
    path('verify-otp/', VerifyOTPView.as_view(), name='verify_otp'),
    path('resend-otp/', ResendOTPView.as_view(), name='resend_otp'),
    path('check-password-strength/', CheckPasswordStrengthView.as_view(), name='check_password_strength'),

```

```

# Password reset URLs
path('password-reset/request/', PasswordResetRequestView.as_view(), name='password_reset_request'),
path('password-reset/verify/', PasswordResetVerifyView.as_view(), name='password_reset_verify'),
path('password-reset/confirm/', PasswordResetConfirmView.as_view(), name='password_reset_confirm'),
]

```

Account/models.py

```

from django.contrib.auth.models import AbstractUser, BaseUserManager
from django.db import models
from django.contrib.auth.models import User
from django.conf import settings
import os
from django.utils import timezone
import random
import string

```

```

class CustomUserManager(BaseUserManager):
    use_in_migrations = True

```

```

    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError('The Email field must be set')
        email = self.normalize_email(email)
        extra_fields.setdefault('username', email) # Set username to email
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

```

```

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        extra_fields.setdefault('is_active', True)
        return self.create_user(email, password, **extra_fields)

```

```

class CustomUser(AbstractUser):
    username = None # Remove username field
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    date_joined = models.DateTimeField(auto_now_add=True)
    is_active = models.BooleanField(default=True)
    is_email_verified = models.BooleanField(default=False)

```

```

# Add related_name arguments to resolve the clash

```

```

groups = models.ManyToManyField(
    'auth.Group',
    verbose_name='groups',
    blank=True,
    related_name='custom_user_set',
    related_query_name='custom_user'
)
user_permissions = models.ManyToManyField(
    'auth.Permission',
    verbose_name='user permissions',
    blank=True,
    related_name='custom_user_set',
    related_query_name='custom_user'
)

```

```

objects = CustomUserManager()

USERNAME_FIELD = 'email'
REQUIRED_FIELDS = ['first_name', 'last_name']

def __str__(self):
    return self.email

class Meta:
    verbose_name = 'User'
    verbose_name_plural = 'Users'

class UserProfile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, related_name='profile')
    bio = models.TextField(blank=True)
    address = models.CharField(max_length=100, blank=True)
    dob = models.DateField(null=True, blank=True)
    phone = models.CharField(max_length=10, blank=True)
    profile_image = models.ImageField(upload_to='profile_pics/', default='default.jpg')
    is_email_verified = models.BooleanField(default=False)

    def __str__(self):
        return f'{self.user.email} Profile'

    def save(self, *args, **kwargs):
        # Create directory if it doesn't exist
        if self.profile_image:
            img_path = os.path.join(settings.MEDIA_ROOT, 'profile_pics')
            os.makedirs(img_path, exist_ok=True)
            super().save(*args, **kwargs)

class EmailVerification(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    otp = models.CharField(max_length=6)
    created_at = models.DateTimeField(auto_now_add=True)
    expires_at = models.DateTimeField()
    is_verified = models.BooleanField(default=False)

    def __str__(self):
        return f'{self.user.email} - {self.otp}'

    def save(self, *args, **kwargs):
        # Set expiration time to 10 minutes from creation if not set
        if not self.expires_at:
            self.expires_at = timezone.now() + timezone.timedelta(minutes=10)
            super().save(*args, **kwargs)

    @staticmethod
    def generate_otp():
        # Generate a 6-digit OTP
        return ''.join(random.choices(string.digits, k=6))

    def is_valid(self):
        # Check if OTP is expired
        return timezone.now() <= self.expires_at and not self.is_verified

class PasswordResetToken(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    token = models.CharField(max_length=100, unique=True)
    created_at = models.DateTimeField(auto_now_add=True)
    expires_at = models.DateTimeField()
    is_used = models.BooleanField(default=False)

    def __str__(self):
        return f'Reset token for {self.user.email}'

    def is_valid(self):
        """Check if token is valid (not expired and not used)"""

```

```
return not self.is_used and timezone.now() < self.expires_at
```

```
@classmethod
def generate_token(cls, user, expiry_hours=24):
    """Generate a password reset token for a user"""
    # Invalidate any existing tokens
    cls.objects.filter(user=user, is_used=False).update(is_used=True)
    # Generate new token
    token = ''.join(random.choices(string.ascii_letters + string.digits, k=32))
    expires_at = timezone.now() + timezone.timedelta(hours=expiry_hours)
    # Create token record
    reset_token = cls.objects.create(
        user=user,
        token=token,
        expires_at=expires_at
    )
    return reset_token
```

Account/serializers.py

```
from Account.models import CustomUser, EmailVerification
from django.contrib.auth.password_validation import validate_password
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer
from rest_framework import serializers
from rest_framework.validators import UniqueValidator
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer
from django.contrib.auth.models import User
from .models import UserProfile, PasswordResetToken
from .utils import validate_password_strength
```

```
class CustomUserSerializer(serializers.ModelSerializer):
    class Meta:
        model = CustomUser
        fields = ('id', 'username', 'email')
```

```
class MyTokenObtainPairSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)

        # These are claims, you can add custom claims
        token['full_name'] = user.profile.full_name
        # token['username'] = user.username
        token['email'] = user.email
        token['bio'] = user.profile.bio
        token['profile_image'] = str(user.profile.profile_image)
        token['verified'] = user.profile.verified
        token['address'] = user.profile.address
        token['dob'] = str(user.profile.dob)
        # ...
        return token
```

```
# Serializer for User model
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['id', 'first_name', 'last_name', 'email']
```

```
# Serializer for UserProfile
class ProfileSerializer(serializers.ModelSerializer):
    user = UserSerializer(read_only=True)
    first_name = serializers.CharField(source='user.first_name', required=False)
    last_name = serializers.CharField(source='user.last_name', required=False)
    email = serializers.EmailField(source='user.email', required=False)
    image_url = serializers.SerializerMethodField()
```

```

class Meta:
    model = UserProfile
fields = ['user', 'first_name', 'last_name', 'email', 'bio', 'profile_image', 'address', 'dob', 'phone', 'image_url']

def get_image_url(self, obj):
    if obj.profile_image and hasattr(obj.profile_image, 'url'):
        request = self.context.get('request')
        if request:
            return request.build_absolute_uri(obj.profile_image.url)
        return obj.profile_image.url
    return '/media/profile_pics/default.jpg'

def validate_profile_image(self, value):
    if value:
        if value.size > 5 * 1024 * 1024: # 5MB limit
            raise serializers.ValidationError("Image size cannot exceed 5MB")
        if not value.content_type.startswith('image/'):
            raise serializers.ValidationError("Invalid image format")
    return value

def update(self, instance, validated_data):
    user_data = validated_data.pop('user', {})
    # Update user fields if present
    for attr, value in user_data.items():
        setattr(instance.user, attr, value)
    instance.user.save()

    # Update profile fields
    for attr, value in validated_data.items():
        setattr(instance, attr, value)
    instance.save()
    return instance

# Register serializer for user registration
class RegisterSerializer(serializers.ModelSerializer):
    password = serializers.CharField(
        write_only=True, required=True, validators=[validate_password])
    password2 = serializers.CharField(write_only=True, required=True)

    class Meta:
        model = CustomUser
        fields = ('email', 'username', 'password', 'password2')

    def validate(self, attrs):
        if attrs['password'] != attrs['password2']:
            raise serializers.ValidationError(
                {"password": "Password fields didn't match."})

        return attrs

    def validate_email(self, value):
        try:
            User.objects.get(email=value)
        except User.DoesNotExist:
            raise serializers.ValidationError("User with this email does not exist")
        return value

    def validate_otp(self, value):
        if value and not value.isdigit():
            raise serializers.ValidationError("OTP must contain only digits")
        return value

# Resend OTP Serializer
class ResendOTPSerializer(serializers.Serializer):

```

```

email = serializers.EmailField(required=True)

def validate_email(self, value):
    try:
        User.objects.get(email=value)
    except User.DoesNotExist:
        raise serializers.ValidationError("User with this email does not exist")
    return value

class PasswordResetRequestSerializer(serializers.Serializer):
    email = serializers.EmailField()

    def validate_email(self, value):
        # Check if user with this email exists
        if not User.objects.filter(email=value).exists():
            raise serializers.ValidationError("User with this email does not exist.")
        return value

class PasswordResetVerifySerializer(serializers.Serializer):
    token = serializers.CharField()

    def validate_token(self, value):
        # Check if token exists and is valid
        try:
            token_obj = PasswordResetToken.objects.get(token=value, is_used=False)
            if not token_obj.is_valid():
                raise serializers.ValidationError("Token has expired.")
        except PasswordResetToken.DoesNotExist:
            raise serializers.ValidationError("Invalid token.")
        return value

class PasswordResetConfirmSerializer(serializers.Serializer):
    token = serializers.CharField()
    password = serializers.CharField(min_length=8, write_only=True)
    password2 = serializers.CharField(min_length=8, write_only=True)

    def validate(self, data):
        # Check if passwords match
        if data['password'] != data['password2']:
            raise serializers.ValidationError("Passwords do not match.")

        # Check if token exists and is valid
        try:
            token_obj = PasswordResetToken.objects.get(token=data['token'], is_used=False)
            if not token_obj.is_valid():
                raise serializers.ValidationError("Token has expired.")
            self.token_obj = token_obj
        except PasswordResetToken.DoesNotExist:
            raise serializers.ValidationError("Invalid token.")

        # Validate password strength
        is_valid, errors = validate_password_strength(data['password'])
        if not is_valid:
            raise serializers.ValidationError({"password": errors})

    return data

    def save(self):
        # Get user from token
        user = self.token_obj.user

        # Set new password
        user.set_password(self.validated_data['password'])
        user.save()

```

```

    # Mark token as used
    self.token_obj.is_used = True
    self.token_obj.save()
    return user

```

frontend/src/pages/login.jsx

```

import React, { useEffect, useState } from 'react';
import { useNavigate, useLocation } from 'react-router-dom';
import axiosInstance from '../utils/axios';
import { Link } from 'react-router-dom';
import { validateLogin } from '../utils/validation';
import useFormValidation from '../utils/useFormValidation';

const Login = () => {
  const navigate = useNavigate();
  const location = useLocation();
  const [successMessage, setSuccessMessage] = useState("");

  // Check for success messages from verification or registration
  useEffect(() => {
    if (location.state?.verified) {
      setSuccessMessage(location.state.message || 'Email verified successfully!');
    } else if (location.state?.registered) {
      setSuccessMessage(location.state.message || 'Registration successful!');
    }
  }, [location]);

  const initialState = {
    email: "",
    password: ""
  };

  const handleLoginSubmit = async (formData) => {
    // JWT expects 'username' and 'password', so map email to username
    const response = await axiosInstance.post('/account/token/', {
      username: formData.email,
      password: formData.password,
    });

    if (response.data.access) {
      localStorage.setItem('access_token', response.data.access);
      localStorage.setItem('refresh_token', response.data.refresh);
      navigate('/');
    }
    return response;
  };

  const {
    formData,
    errors,
    serverError,
    isSubmitting,
    handleChange,
    handleBlur,
    handleSubmit,
    setServerError
  } = useFormValidation(initialState, validateLogin, handleLoginSubmit);

  // Handle case where login fails due to email not verified
  useEffect(() => {
    if (serverError && serverError.includes('Email not verified')) {

```

```

    // Redirect to verification page
    navigate('/verify-otp', { state: { email: formData.email } });
  }
}, [serverError, navigate, formData.email]);

return (
  <div className="min-h-screen flex items-center justify-center bg-
[url('/src/assets/Signup_background_image.jpg')] bg-cover">
    <div className="w-full max-w-6xl bg-transparent flex flex-col md:flex-row rounded-2xl overflow-hidden
">
      { /* Left Side */}
      <div className="md:w-1/2 w-full bg-transparent px-6 md:px-10 py-10 text-white flex flex-col justify-
between">
        <h1 className="text-4xl md:text-6xl font-bold leading-tight mb-6">Sign<br />In.</h1>
        <p className="text-base text-white/80 mb-10 hidden md:block">
          Beyond identification, Plantify offers an AI assistant that not only names your plant but also provides
          guidance on care, warns about potential diseases, and suggests treatment options. This interactive experience sets
          Plantify apart, making it a valuable tool for plant enthusiasts.
        </p>
        <p className="text-sm text-white/60">© Plantify-AI</p>
      </div>

      { /* Right Side */}
      <div className="md:w-1/2 w-full bg-green-100 px-6 md:px-10 py-6 md:py-3 rounded-none md:rounded-
l-3xl relative flex flex-col justify-center">
        <h2 className="text-2xl font-semibold mb-2 text-green-500 text-center md:text-left">Sign In</h2>

        {successMessage && (
          <div className="bg-green-100 border border-green-400 text-green-700 px-4 py-3 rounded relative mb-
4" role="alert">
            <span className="block sm:inline">{successMessage}</span>
          </div>
        )}

        <form className="mt-8 space-y-6" onSubmit={handleSubmit}>
          <div className="rounded-md shadow-sm -space-y-px">
            <div className="mb-4">
              <label htmlFor="email" className="block text-gray-700">Email address</label>
              <input
                id="email"
                name="email"
                type="email"
                required
                className={`w-full px-4 py-2 border ${errors.email ? 'border-red-500' : 'border-gray-400'} rounded-
md focus:outline-none focus:ring-1 focus:ring-gray-400`}
                placeholder="Email address"
                value={formData.email}
                onChange={handleChange}
                onBlur={handleBlur}
              />
              {errors.email && <p className="text-red-500 text-xs mt-1">{errors.email}</p>}
            </div>
            <div className="mb-4">
              <label htmlFor="password" className="block text-gray-700">Password</label>
              <input
                id="password"
                name="password"
                type="password"
                required
                className={`w-full px-4 py-2 border ${errors.password ? 'border-red-500' : 'border-gray-400'}
rounded-md focus:outline-none focus:ring-1 focus:ring-gray-400`}
                placeholder="Password"
                value={formData.password}
                onChange={handleChange}

```



```

        onBlur={handleBlur}
      />
      {errors.password && <p className="text-red-500 text-xs mt-1">{ errors.password}</p>
      <div className="flex justify-end mt-1">
        <Link to="/forgot-password" className="text-sm text-green-500 hover:text-green-700">
          Forgot password?
        </Link>
      </div>
    </div>
  </div>
</div>

{serverError && !serverError.includes('Email not verified') && (
  <div className="bg-red-100 border border-red-400 text-red-700 px-4 py-3 rounded relative"
role="alert">
    <span className="block sm:inline">{serverError}</span>
  </div>
)}

<div>
  <button
    type="submit"
    disabled={isSubmitting}
    className="w-full bg-green-500 text-white py-2 rounded-md hover:bg-green-600 transition duration-
300"
  >
    {isSubmitting ? 'Signing in...' : 'Sign in'}
  </button>
  <p className='mt-2 text-center md:text-left'>Don't have an account? <Link to="/signup"
className="text-green-500">Sign Up</Link></p>
</div>
</form>
</div>
</div>
</div>
);
};

```

export default Login;

frontend/src/pages/signup.jsx

```

import React from 'react';
import { useNavigate } from 'react-router-dom';
import axiosInstance from '../utils/axios';
import { Link } from 'react-router-dom';
import { validateSignup } from '../utils/validation';
import useFormValidation from '../utils/useFormValidation';

const Signup = () => {
  const navigate = useNavigate();
  const initialState = {
    email: "",
    password: "",
    password2: "",
    first_name: "",
    last_name: ""
  };

  const handleSignupSubmit = async (formData) => {
    const response = await axiosInstance.post('/account/register/', formData);
    if (response.data.success) {
      // Check if email verification is required
    }
  }

```

```

    if (response.data.data?.requires_verification) {
      // Redirect to OTP verification page
      navigate('/verify-otp', {
        state: { email: formData.email }
      });
    } else {
      // Directly redirect to login
      navigate('/login', {
        state: {
          registered: true,
          message: 'Registration successful! You can now log in.'
        }
      });
    }
  }
  return response;
};

const {
  formData,
  errors,
  serverError,
  isSubmitting,
  handleChange,
  handleBlur,
  handleSubmit
} = useFormValidation(initialState, validateSignup, handleSignupSubmit);

return (
  <div className="min-h-screen flex items-center justify-center bg-
[url('./src/assets/Signup_background_image.jpg')] bg-cover">
    <div className="w-full max-w-6xl bg-transparent flex flex-col md:flex-row rounded-2xl overflow-hidden">
      /* Left Side */
      <div className="md:w-1/2 w-full bg-transparent px-6 md:px-10 py-10 text-white">
        <h1 className="text-4xl md:text-6xl font-bold leading-tight mb-6">Sign<br />Up.</h1>
        <p className="text-base text-white/80 mb-10 hidden md:block">
          Beyond identification, Plantify offers an AI assistant that not only names your plant but also provides
          guidance on care, warns about potential diseases, and suggests treatment options. This interactive experience sets
          Plantify apart, making it a valuable tool for plant enthusiasts.
        </p>
        <p className="text-sm text-white/60">© Plantify-AI</p>
      </div>

      /* Right Side */
      <div className="md:w-1/2 w-full bg-green-100 px-6 md:px-10 py-6 md:py-3 rounded-none md:rounded-
l-3xl relative">
        <h2 className="text-2xl font-semibold mb-2 text-green-500 text-center md:text-left">Sign Up</h2>
        <form className="mt-8 space-y-6" onSubmit={handleSubmit}>
          {serverError && (
            <div className="bg-red-100 border border-red-400 text-red-700 px-4 py-3 rounded relative"
            role="alert">
              <span className="block sm:inline">{serverError}</span>
            </div>
          )}
          <div className="rounded-md shadow-sm -space-y-px">
            <div className="mb-4">
              <label htmlFor="first_name" className="block text-gray-700">First Name</label>
              <input
                id="first_name"
                name="first_name"
                type="text"
                required
                className={`w-full px-4 py-2 border ${errors.first_name ? 'border-red-500' : 'border-gray-400'}

```

```

rounded-md focus:outline-none focus:ring-1 focus:ring-gray-400` }
    placeholder="First Name"
    value={ formData.first_name }
    onChange={ handleChange }
    onBlur={ handleBlur }
  />
  {errors.first_name && <p className="text-red-500 text-xs mt-1">{errors.first_name}</p>}
</div>
<div className="mb-4">
  <label htmlFor="last_name" className="block text-gray-700">Last Name</label>
  <input
    id="last_name"
    name="last_name"
    type="text"
    required
    className={`w-full px-4 py-2 border ${errors.last_name ? 'border-red-500' : 'border-gray-400'}
rounded-md focus:outline-none focus:ring-1 focus:ring-gray-400` }
    placeholder="Last Name"
    value={ formData.last_name }
    onChange={ handleChange }
    onBlur={ handleBlur }
  />
  {errors.last_name && <p className="text-red-500 text-xs mt-1">{errors.last_name}</p>}
</div>
<div className="mb-4">
  <label className="block text-gray-700">Email </label>
  <input
    id="email"
    name="email"
    type="email"
    required
    className={`w-full px-4 py-2 border ${errors.email ? 'border-red-500' : 'border-gray-400'} rounded-
md focus:outline-none focus:ring-1 focus:ring-gray-400` }
    placeholder="Email address"
    value={ formData.email }
    onChange={ handleChange }
    onBlur={ handleBlur }
  />
  {errors.email && <p className="text-red-500 text-xs mt-1">{errors.email}</p>}
</div>
<div className="mb-4">
  <label htmlFor="password" className="block text-gray-700">Password</label>
  <input
    id="password"
    name="password"
    type="password"
    required
    className={`w-full px-4 py-2 border ${errors.password ? 'border-red-500' : 'border-gray-400'}
rounded-md focus:outline-none focus:ring-1 focus:ring-gray-400` }
    placeholder="Password"
    value={ formData.password }
    onChange={ handleChange }
    onBlur={ handleBlur }
  />
  {errors.password && <p className="text-red-500 text-xs mt-1">{errors.password}</p>}
</div>
<div className="mb-4">
  <label htmlFor="password2" className="block text-gray-700">Confirm Password</label>
  <input
    id="password2"
    name="password2"
    type="password"
    required
    className={`w-full px-4 py-2 border ${errors.password2 ? 'border-red-500' : 'border-gray-400'}
rounded-md focus:outline-none focus:ring-1 focus:ring-gray-400` }

```

CHAPTER-10

OUTPUT SCREEN

CHAPTER 10

RESULT & OUTPUT SCREENS

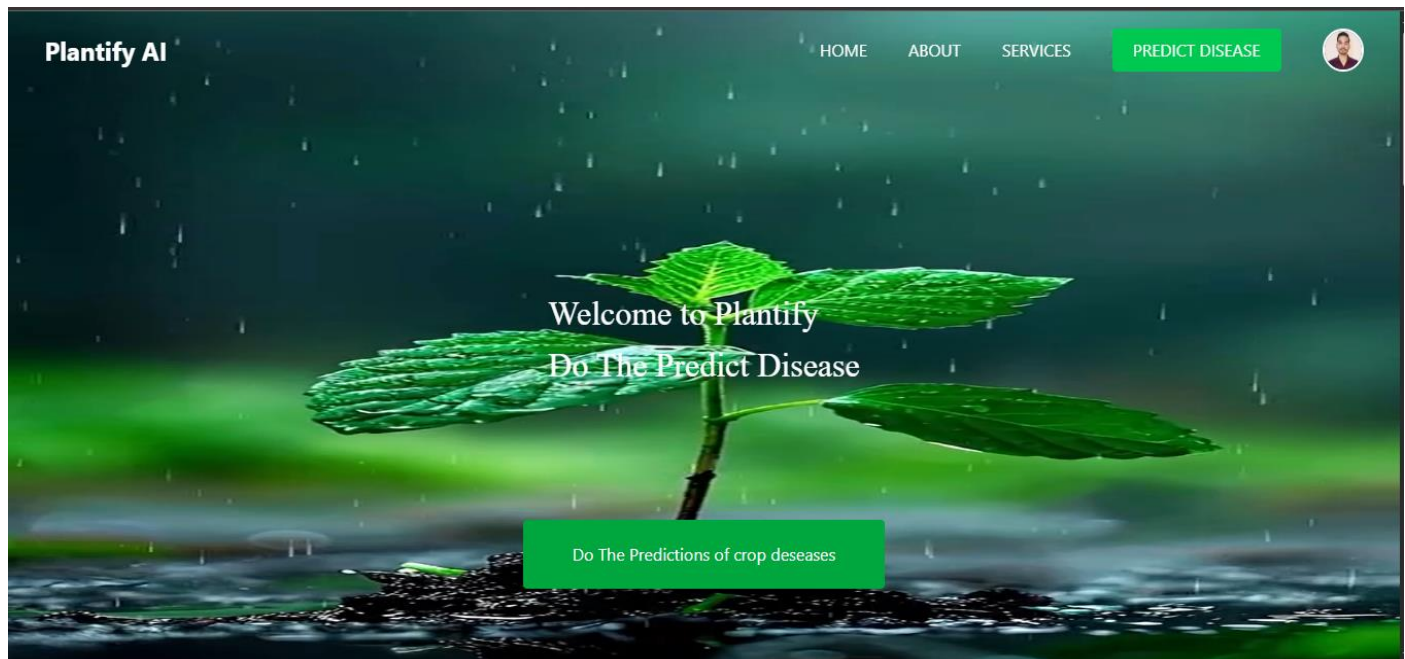


Figure 10.1 Home Page

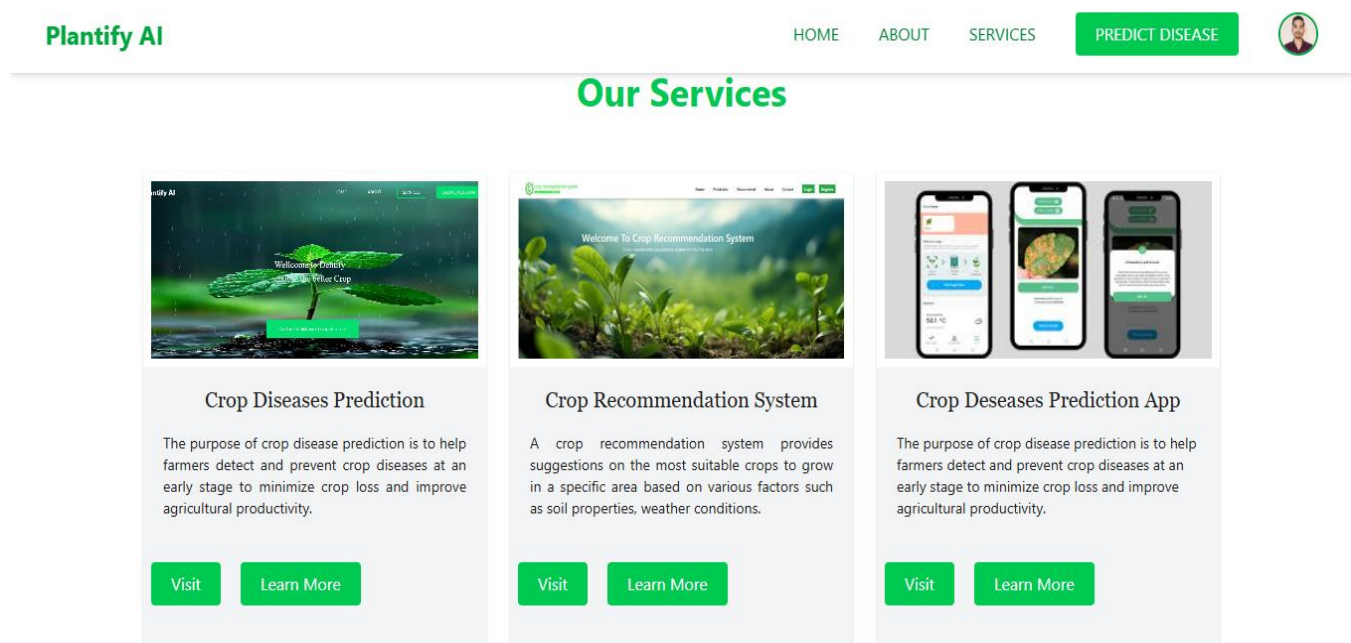
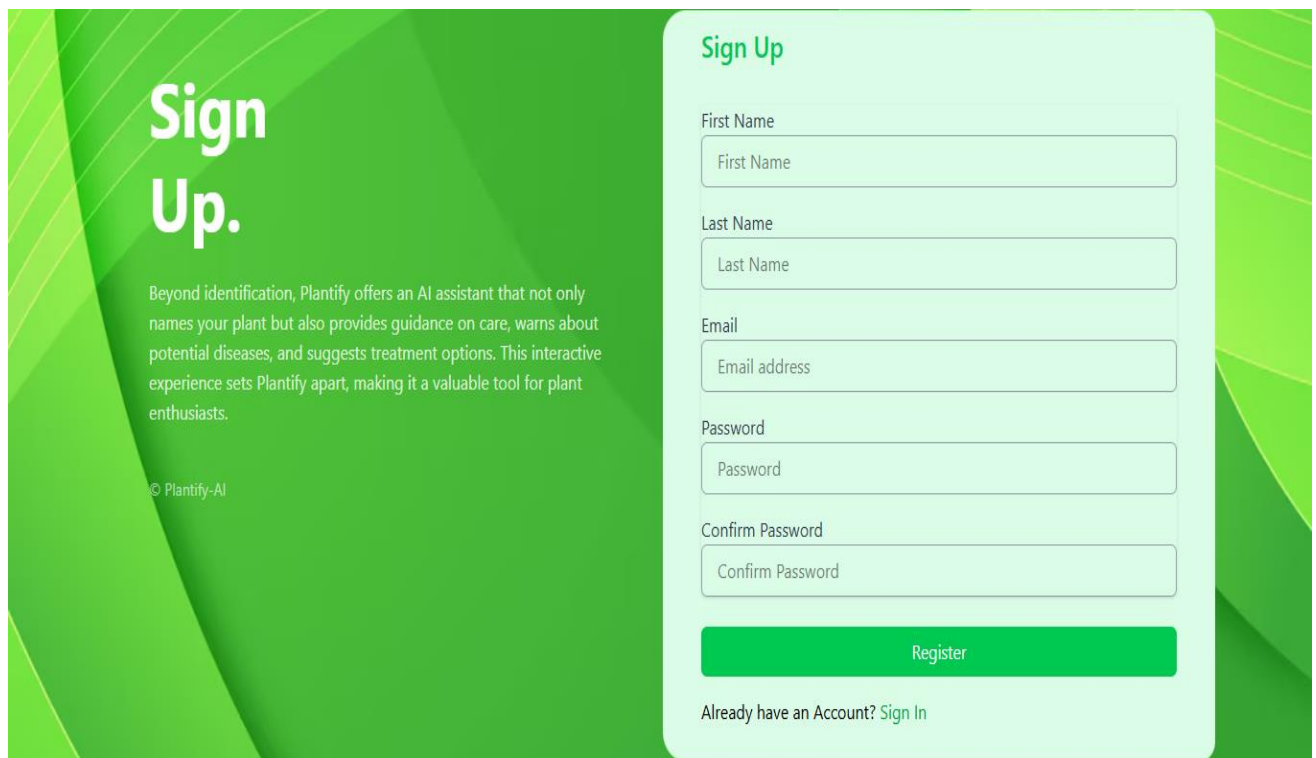


Figure 10.2 Service Page



The Sign Up page features a green background with abstract leaf patterns. On the left, the text 'Sign Up.' is displayed in large white font. Below it, a paragraph describes the AI assistant's capabilities. A copyright notice '© Plantify-AI' is at the bottom left. On the right, a white rounded rectangle contains the sign-up form. The form includes fields for First Name, Last Name, Email, Password, and Confirm Password, each with a placeholder text. A green 'Register' button is at the bottom of the form. Below the button, a link 'Already have an Account? Sign In' is provided.

Sign Up.

Beyond identification, Plantify offers an AI assistant that not only names your plant but also provides guidance on care, warns about potential diseases, and suggests treatment options. This interactive experience sets Plantify apart, making it a valuable tool for plant enthusiasts.

© Plantify-AI

Sign Up

First Name
First Name

Last Name
Last Name

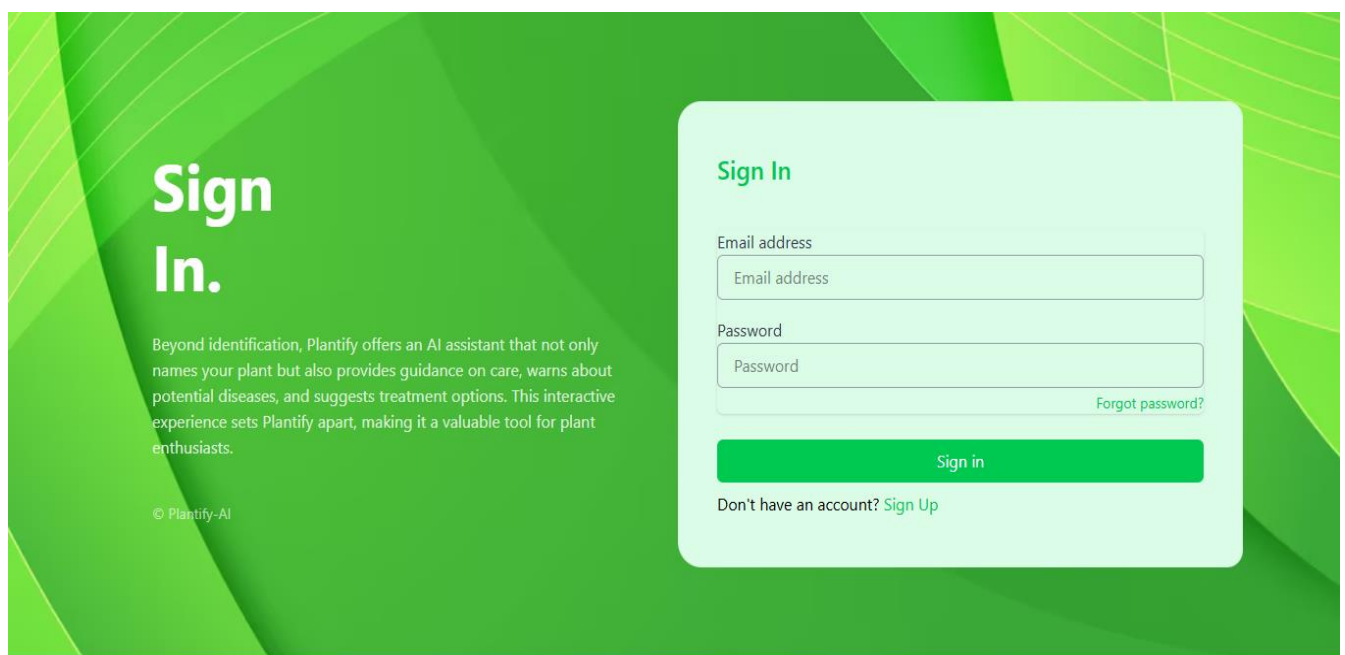
Email
Email address

Password
Password

Confirm Password
Confirm Password

Register

Already have an Account? [Sign In](#)

Figure 10.3 Signup Page

The Sign In page features a green background with abstract leaf patterns. On the left, the text 'Sign In.' is displayed in large white font. Below it, a paragraph describes the AI assistant's capabilities. A copyright notice '© Plantify-AI' is at the bottom left. On the right, a white rounded rectangle contains the sign-in form. The form includes fields for Email address and Password, each with a placeholder text. A 'Forgot password?' link is located to the right of the Password field. A green 'Sign in' button is at the bottom of the form. Below the button, a link 'Don't have an account? Sign Up' is provided.

Sign In.

Beyond identification, Plantify offers an AI assistant that not only names your plant but also provides guidance on care, warns about potential diseases, and suggests treatment options. This interactive experience sets Plantify apart, making it a valuable tool for plant enthusiasts.

© Plantify-AI

Sign In

Email address
Email address

Password
Password

[Forgot password?](#)

Sign in

Don't have an account? [Sign Up](#)

Figure 10.4 Login Page

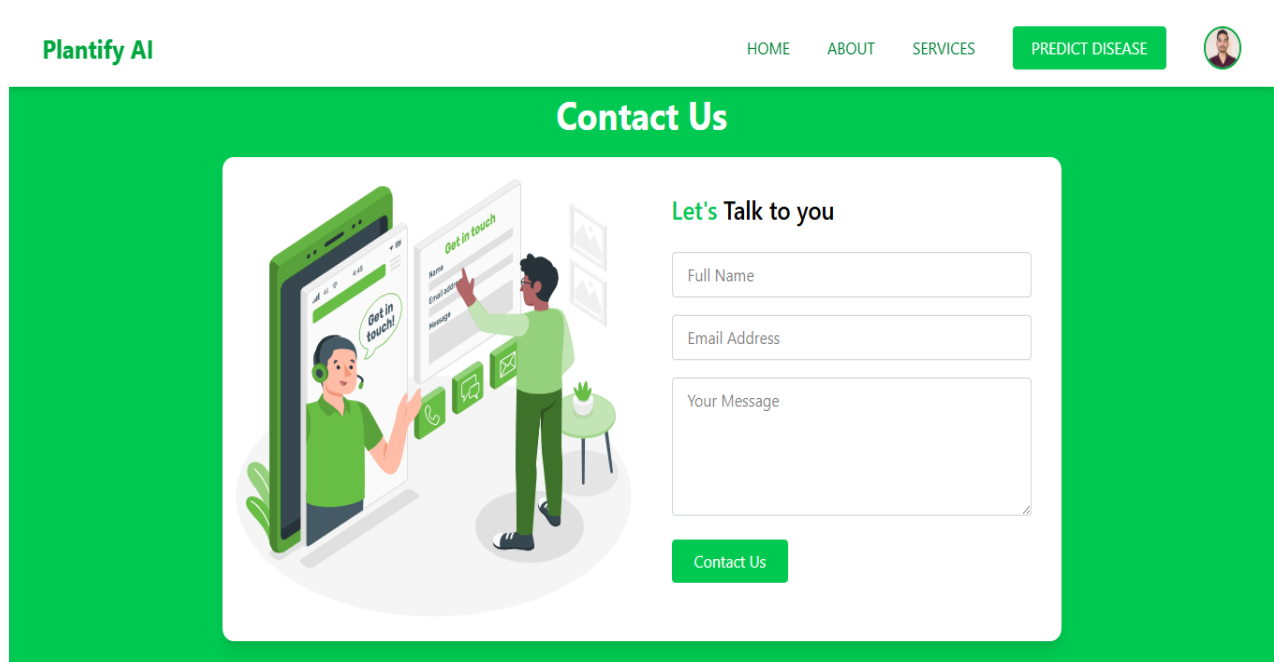


Figure 10.5 Contact Page

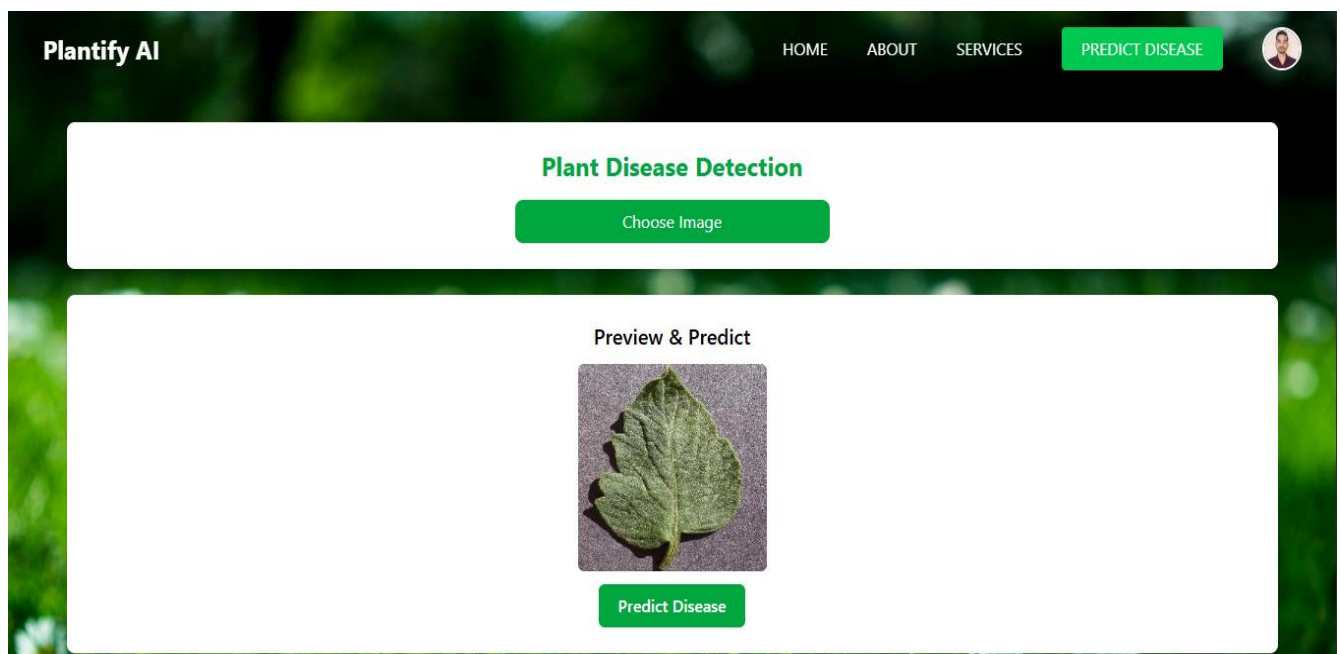


Figure 10.6 Dashboard Page

Chapter 11

Future work

CHAPTER 11

CONCLUSION & FUTURE WORK

11.1.CONCLUSION

The Crop Disease Prediction System leverages deep learning techniques, specifically Convolutional Neural Networks (CNNs), to accurately classify plant diseases from leaf images. This project showcases the transformative potential of AI in agriculture by providing an automated and efficient solution for early disease detection. The system features an intuitive user interface that allows farmers and agronomists to upload crop images, receive disease predictions, and access detailed results seamlessly. With a robust CNN-powered backend, the system ensures reliable performance, aiming to support agricultural professionals in enhancing crop health and yield management.

FUTURE WORK

- **Integration of Additional Crop Diseases:** Expand the model's capabilities to recognize a wider variety of plant diseases across different crop types for more comprehensive agricultural diagnostics.
- **Real-Time Prediction:** Develop and integrate a real-time image analysis feature to enable instant disease detection in the field, enhancing system responsiveness and usability.
- **Mobile Application:** Create a mobile app to increase accessibility for farmers and field workers, allowing them to capture and analyse crop images directly from their smartphones.
- **Enhanced Dataset:** Incorporate larger, more diverse, and region-specific agricultural datasets to improve model accuracy, generalization, and reduce prediction bias.
- **Localization Feature:** Enable the system to identify and highlight the exact location of disease symptoms on the crop image for more precise and actionable insights.
- **Multi-Language Support:** Implement support for multiple languages to ensure the system is accessible to users from different regions and linguistic backgrounds.
- **Cloud Integration:** Integrate cloud-based infrastructure for secure data storage and scalable processing, improving performance, collaboration, and remote accessibility.

REFERENCES

1. RAM Santhosh, A. S. Laxmi, S. A. Chepuri, V. Santosh, V. Kothareddy, and S. Chopra, "Review and further prospects of plant disease detection using machine learning," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 7, pp. 105-115, 2021.
2. B. V. Gokulnath and G. Usha Devi, "A survey on plant disease prediction using machine learning and deep learning techniques," *Intelligence Artificial*, vol. 23, pp. 136-154, 2020.
3. J. Liu and X. Wang, "Plant diseases and pests detection based on deep learning: A review," *Plant Methods*, vol. 17, p. 22, 2021.
4. T. Su, W. Li, P. Wang, and C. Ma, "Dynamics of peroxisome homeostasis and its role in stress response and signaling in plants," *Frontiers in Plant Science*, vol. 10, p. 705, 2019.
5. Ayaz, Muhammad, et al. "Internet-of-Things (IoT)-based smart agriculture: Toward making the fields talk." *IEEE access* 7 (2019): 129551-129583.NCBI :
6. S. Roy, R. Ray, S. R. Dash, and M. K. Giri, "Plant disease detection using machine learning tools with an overview on dimensionality reduction," in *Data Analytics in Bioinformatics*, R. Satpathy, T. Choudhury, S. Satpathy, S. N. Mohanty, and X. Zhang, Eds. Beverly, MA: Scrivener Publishing LLC, 2021, pp. 109-144.
7. D. B. Collinge, H. J. L. Jørgensen, M. A. C. Latz, A. Manzotti, F. Ntana, E. C. Rojas, et al., "Searching for novel fungal biological control agents for plant disease control among endophytes," in *Endophytes for a Growing World*, B. R. Murphy, F. M. Doohan, M. J. Saunders, and T. R. Hodkinson, Eds. Cambridge: Cambridge University Press, 2019, pp. 25-51.
8. Celia A Harvey, Zo Lalaina Rakotobe, Nalini S Rao, Radhika Dave, HeryRazafimahatratra, Revo Hasinandrianina Rabarijohn, Hagino Rajaofara, and James L MacKinnon. Extreme vulnerability of smallholder farmers to agricultural risks and climate change in Madagascar. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 369(1639):20130089, 2014.
9. Mohanty, Sharada, David P., Marcel, and Hughes. Using deep learning for image-based plant disease detection, Sep 2016.

10. Andreas Kamilaris, Francesc X. Prenafeta-Boldu“Deep learning in agriculture: A survey”,
Computers and Electronics in Agriculture 147,70–90, 2018.
11. Ferentinos, K.P. Deep learning models for plant disease detection and diagnosis. Compute.
Electron. Agric. 2018, 145, 311–318. [Google Scholar] [CrossRef].
12. Dataset : <https://www.kaggle.com/datasets/vipooooool/new-plant-diseases-dataset>
13. GitHub Repository Link : <https://github.com/amansahu47388/PlantifyAI>
14. Django Documentation : <https://docs.djangoproject.com/en/5.1/>

APPENDIX-1

GLOSSARY OF TERMS

Accuracy

A measure of how correct the predictions or classifications made by a machine learning model are.

API (Application Programming Interface)

A set of tools and protocols that allow different software components to communicate and interact.

AI (Artificial Intelligence)

The simulation of human intelligence processes by machines, particularly computer systems.

Backend

The server-side part of an application, handling business logic, database interactions, and integrations.

Bug

An error or flaw in software that causes it to produce incorrect or unexpected results.

CLI (Command Line Interface)

A text-based interface is used to interact with software or an operating system.

CNN (Convolutional Neural Network)

A type of deep learning model used for image recognition and classification tasks.

CSS (Cascading Style Sheets)

A language used to style the layout and design of webpages.

Database

An organized collection of data stored and accessed electronically.

Deployment

The process of making a software application available for use.

Django

A high-level Python web framework used for rapid development and clean design.

Frontend

The client-side part of an application, including the user interface and experience.

GUI (Graphical User Interface)

A visual interface allowing users to interact with a program through graphical elements.

HTML (Hypertext Markup Language)

The standard language is used to create and structure webpages.

Deep Learning

Deep Learning is a subfield of machine learning where models, especially neural networks with many layers, learn to perform tasks by training on large datasets, without manual feature engineering

Middleware

Software that connects different components or applications, facilitating communication and data management.

MVC (Model-View-Controller)

A software design pattern for developing applications with separated concerns.

Non-Functional Requirements

Quality attributes such as performance, security, and scalability, that define how a system operates.

Python

A versatile, high-level programming language used for backend development and Deep learning.

Scalability

The ability of a system to handle increased load or expand its capacity seamlessly.

Static Files

Resources like CSS, JavaScript, and images used by web applications to enhance the interface and functionality.

TensorFlow

An open-source library is used for machine learning and deep learning tasks.

URL (Uniform Resource Locator)

The address of a resource on the internet.

Usability

The ease with which users can interact with a system or application.

Validation

The process of ensuring that a system meets the required specifications and standards.

Verification

Checking that the system meets the design and implementation specifications.

Views

The presentation layer in the MVC architecture that displays information to the user and handles user interactions.

PROJECT SUMMARY

ABOUT PROJECT

Title of the project	Design and Development of Crop Disease Prediction System
Semester	6th Semester
Members	3 Members
Team Leader	Aman Sahu
Describe role of every member in the project	<p>1. Aman Sahu - Project Leader:- Aman has overseen the entire project and model training, ensured timely completion, and coordinated team efforts.</p> <p>2. Monu Kushwah - Technical Lead:- Monu has designed the technical architecture of the application and DL Model, led coding efforts, and ensured the project's technical quality.</p> <p>3. Shyam Sahu - UI/UX Designer:- Shyam has created visually appealing and user-friendly interfaces, designed the user experience, and ensured consistency across the project.</p>
What is the motivation for selecting this project?	The Crop Disease Prediction System, based on deep learning, aims to enhance early detection of plant diseases and improve accessibility to timely and affordable agricultural support, addressing real-world farming challenges and improving crop health and yield outcomes.
Project Type (Desktop Application, Web Application, Mobile App, Web)	Web Application

Tools & Technologies

Programming language used	Python, JavaScript
Compiler used (with version)	Python 3.10, Chrome v8 engine

IDE used (with version)	Visual Studio Code
Front End Technologies (with version, wherever Applicable)	HTML5 CSS3 JavaScript Tailwind CSS
Back End Technologies (with version, wherever applicable)	Django 5.1.1 django-cors-headers 4.4.0 django-storages 1.14.4 django-rest-framework 3.15.2
Database used (with version)	Sqlite-3

SOFTWARE DESIGN & CODING

Is prototype of the software developed?	Yes
SDLC model followed (Waterfall, Agile, Spiral etc.)	Iterative & Incremental Model
Why above SDLC model is followed?	The Iterative SDLC Model ensures that the project is flexible, efficient, and robust, making it the ideal choice for a Deep learning-based classification system like this one.
Justify that the SDLC model mentioned above is followed in the project.	The Iterative and Incremental SDLC Model is the most suitable approach for the Crop Disease Prediction System , as it provides the flexibility to adapt and refine both the deep learning model and system components throughout the project. With deep learning applications requiring frequent adjustments, constant feedback, and testing, the iterative approach ensured that the system evolved progressively, leading to a highly optimized and robust solution.
Software Design approach followed (Functional object-oriented)	Object-Oriented Design (OOD) was followed for the development of the Crop Disease Prediction System . This approach helped in organizing the system into distinct objects that represent real-world entities, making the design modular, flexible, and maintainable.
Name the diagrams developed (according to the Design approach followed)	Flow Chart Use Case Diagram Data Flow Diagram

In case Object Oriented approach is followed, which of the OOPS principles are covered in design?	Class Object Encapsulation Modularity Inheritance
No. of Tiers (example 3-tier)	03
Total no. of front-end pages	05
Total no. of tables in database	02
Database is in which Normal Form?	2 Normal Form
Are the entries in the database encrypted?	No
Front end validations applied (Yes / No)	Yes
Session management done (in case of web applications)	Yes
Is application browser compatible (in case of web applications)	Yes
Exception handling done (Yes / No)	Yes
Commenting done in code (Yes / No)	Yes
Naming convention followed (Yes / No)	Yes
Total no. of Use-cases	10
Give titles of Use-cases	Signup, Verification, Login, View Profile, Edit Profile, Dashboard, Make Prediction, Predicted Result, Contact, Create Users, View Users.

Project Requirements

MVC architecture followed (Yes / No)	No
If yes, write the name of MVC architecture followed (MVC-1, MVC-2)	
Design Pattern used (Yes / No)	No
If yes, write the name of Design Pattern used	
Interface type (CLI / GUI)	GUI (Graphical User Interface)
No. of Actors	02
Name of Actors	Users (General User), Admin
Total no. of Functional Requirements	09
List few important non-Functional Requirements	System Availability Scalability Data Security Response Time

Testing

Which testing is performed? (Manual or Automation)	Manual
Is Beta testing done for this project?	No

Write project narrative covering above mentioned points

Project Narrative: Crop Disease Prediction System

The Crop Disease Prediction System (CDPS) is an innovative solution leveraging deep learning to advance agricultural healthcare. This project was developed to address critical challenges in modern farming, such as delayed disease diagnosis, reduced crop yields, and limited access to expert agricultural support—especially in rural and underserved farming communities.

CDPS focuses on the early detection and classification of crop diseases, enabling farmers to take timely and informed actions to protect their crops. By utilizing cutting-edge deep learning techniques in image processing and computer vision, the system ensures high accuracy and reliability in identifying plant health issues.

The project aligns with the goal of improving accessibility to affordable and effective agricultural support, offering a user-friendly diagnostic tool that reduces dependency on traditional, often costly, expert consultations. It serves as a preventive agricultural measure, promoting healthier crops and reducing the risk of widespread infestations and crop failure.

With a strong foundation in AI and full-stack development, CDPS combines advanced technology with a commitment to solving real-world agricultural problems. Its societal impact lies in increasing awareness, democratizing plant healthcare, and contributing to global food security and sustainable farming practices.

Aman Sahu 0187AS221007

Guide Signature
Dr. Vasima Khan
Head & Asso. Prof.
Department of CSE-AI&DS

Shyam Sahu 0187AS221059

Monu Kushwah 0187AS221026