# Crop Health Analysis Using NDVI

## Project Purpose & Objectives:

The Crop Health Analysis Using NDVI system aims to provide farmers with a powerful, AI-driven tool for real-time crop health monitoring. By utilizing advanced satellite imagery and machine learning models, the system enables farmers to assess the condition of their crops more accurately and efficiently than traditional methods. This tool allows farmers to monitor crop health based on NDVI (Normalized Difference Vegetation Index) derived from satellite images, providing real-time insights for informed decisions on irrigation, fertilization, pest control, and overall crop management.

The system highlights plant stress by measuring differences in light absorption and reflection. Healthier plants reflect more near-infrared light and absorb more red light, while stressed or unhealthy plants show the opposite pattern.

By identifying low NDVI areas in real time:Irrigation: Farmers can detect drought stress early and irrigate only where needed, reducing water waste.Fertilization: Nutrient-deficient areas show lower NDVI values. Farmers can target these zones with precise fertilizer application, saving cost and improving efficiency.

Pest Control: Sudden drops in NDVI may signal pest damage. Early detection allows for localized pesticide use, minimizing environmental impact. So, NDVI-based analysis gives farmers a visual, data-driven way to take quick, targeted actions—boosting productivity and reducing input costs.

This project is supported by a FastAPI-based backend for a crop health classification application. It allows users to upload images (either RGB or NDVI), classify them as **dead - unhealthy- moderate - healthy**,  using a pre-trained Inception model, and manage user authentication and image storage seamlessly. The backend integrates with Supabase for database operations and secure file storage, ensuring that farmers and other stakeholders can reliably access, review, and act upon crop health data through a modern, scalable API.

## Dataset Overview

### Data Source & Access

- Retrieved from NASA's LAADS archive (MODIS Terra MOD13QA product)

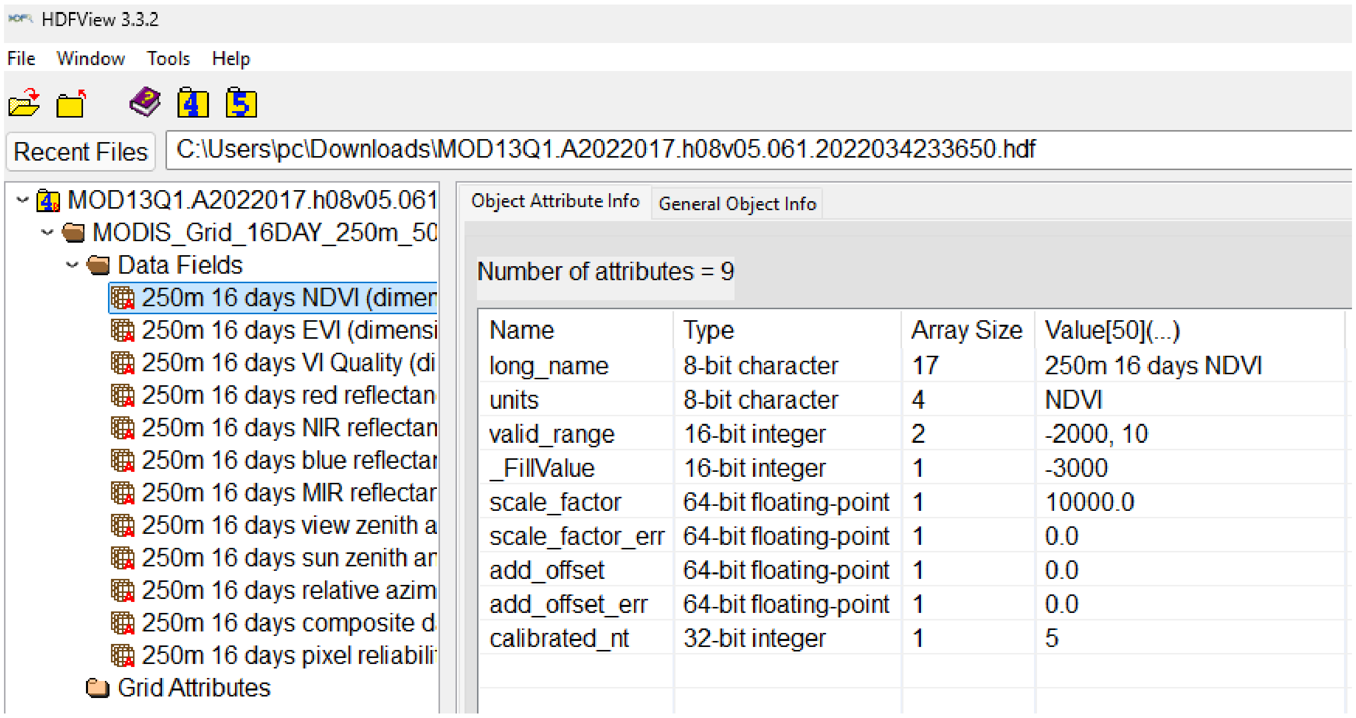
### Spatial & Temporal Resolution

- Spatial: 250 m × 250 m per pixel, suitable for field-scale vegetation monitoring.
- Temporal: 16-day composites, balancing revisit frequency with data volume.

### File Format & Structure

Structure:

- Format: HDF4 files, each covering a 4 800 × 4 800 pixel tile.
- Raw Values: Stored as integers multiplied by 10 000.
- Fill Value: –3000 signals invalid or missing data.



C:\Users\pc\Downloads\MOD13Q1.A2022017.h08v05.061.2022034233650.hdf

MOD13Q1.A2022017.h08v05.061
MODIS_Grid_16DAY_250m_50
Data Fields
250m 16 days NDVI (dimer
250m 16 days EVI (dimensi
250m 16 days VI Quality (di
250m 16 days red reflectan
250m 16 days NIR reflectan
250m 16 days blue reflectar
250m 16 days MIR reflectar
250m 16 days view zenith a
250m 16 days sun zenith ar
250m 16 days relative azim
250m 16 days composite d
250m 16 days pixel reliabili
Grid Attributes

Number of attributes = 9

| Name | Type | Array Size | Value[50](...) |
|---|---|---|---|
| long_name | 8-bit character | 17 | 250m 16 days NDVI |
| units | 8-bit character | 4 | NDVI |
| valid_range | 16-bit integer | 2 | -2000, 10 |
| _FillValue | 16-bit integer | 1 | -3000 |
| scale_factor | 64-bit floating-point | 1 | 10000.0 |
| scale_factor_err | 64-bit floating-point | 1 | 0.0 |
| add_offset | 64-bit floating-point | 1 | 0.0 |
| add_offset_err | 64-bit floating-point | 1 | 0.0 |
| calibrated_nt | 32-bit integer | 1 | 5 |

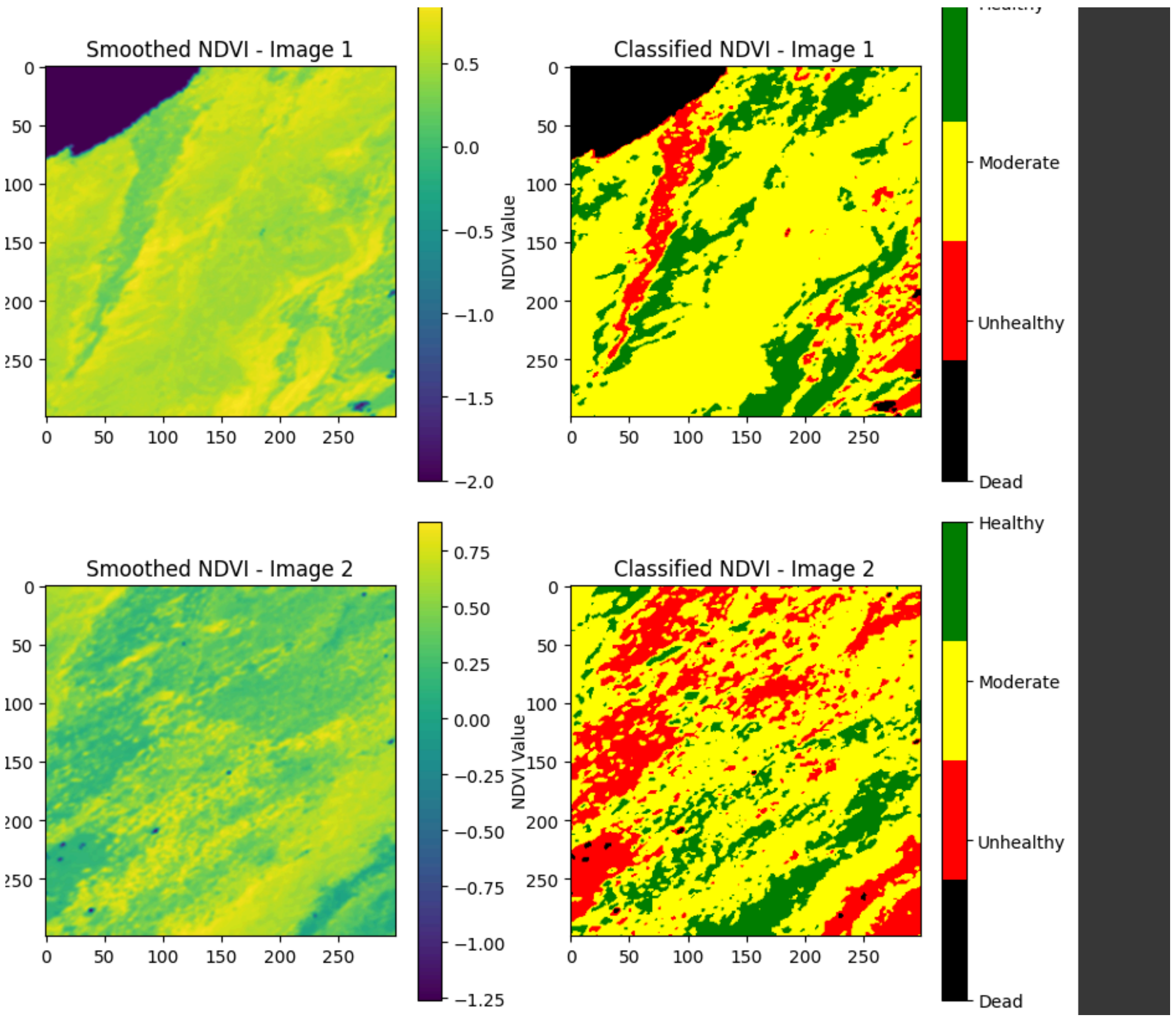
## Implementation Steps

### Automated Data Acquisition:.

- Authentication: Connected to NASA's LAADS DAAC archive using a Bearer token for secure programmatic access.
- Download Automation: Used wget with recursive mirroring and robotic restrictions disabled to fetch HDF4 datasets directly from the archived order link.
- Storage Strategy: Files were saved to a shared Google Drive directory to enable collaborative access across team members.
- Cloud Filtering: Leveraged the MOD13QA quality assurance band to identify and mask out pixels affected by clouds improving NDVI accuracy.

### Data Processing Pipeline:

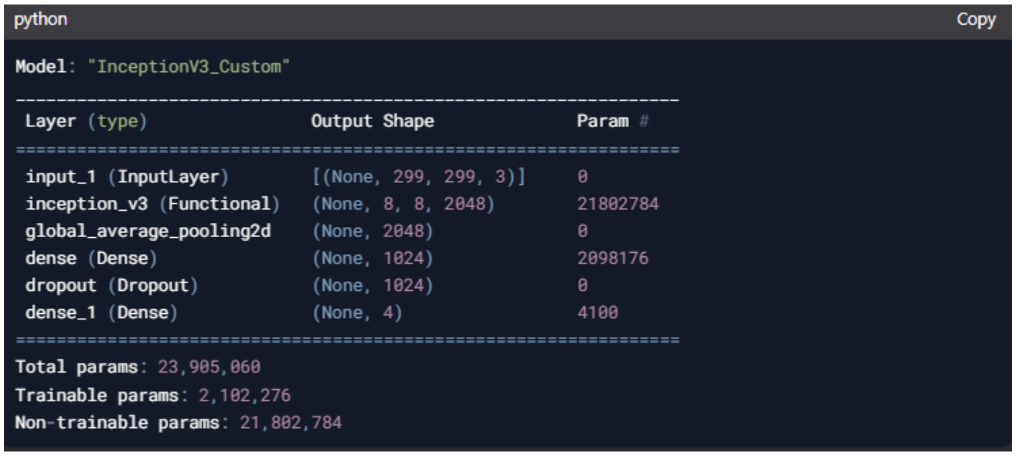
- Chunking: Split each 4,800 × 4,800 tile into manageable 500 × 500.
- Fill Value Handling: Filtered out pixels with the default MODIS fill value (–3000) to prevent contamination of input data.
- Normalization: Scaled NDVI values by dividing by 10,000 to restore the original float range of [–1.0, 1.0].
- Resizing: Resized chunks to 299 × 299 to match the expected input shape for InceptionV3.
- Smoothing: Applied Gaussian blur to reduce noise while preserving the spatial integrity of vegetation zones.
- Mean Imputation: Replaced any NaN or masked-out values with the mean of valid pixels within the same patch, avoiding model disruptions.
- Data Augmentation
  - We experimented with standard augmentation techniques (rotation, shifting, zooming, flipping) using ImageDataGenerator.
  - However, despite a minor lift in validation accuracy, the additional training time and compute demands were not worthwhile given our hardware limitations.
  - As a result, the augmentation steps were commented out in the final version of the pipeline and are not used in production.


Smoothed NDVI - Image 1
Classified NDVI - Image 1
Smoothed NDVI - Image 2
Classified NDVI - Image 2
NDVI Value
Healthy
Moderate
Unhealthy
Dead

## Model Training:

- Model Architecture (InceptionV3)
    - Base Model: InceptionV3 with ImageNet weights (frozen)
    - Optimizer: Adam (lr=0.0001)
    - Loss: Categorical Crossentropy

### Model Summary:



```
Model: "InceptionV3_Custom"
_________________________________________________________________
 Layer (type)                 Output Shape             Param #
=================================================================
 input_1 (InputLayer)         [(None, 299, 299, 3)]    0
 inception_v3 (Functional)    (None, 8, 8, 2048)       21802784
 global_average_pooling2d     (None, 2048)             0
 dense (Dense)                (None, 1024)             2098176
 dropout (Dropout)            (None, 1024)             0
 dense_1 (Dense)              (None, 4)                4100
=================================================================
Total params: 23,905,060
Trainable params: 2,102,276
Non-trainable params: 21,802,784
```

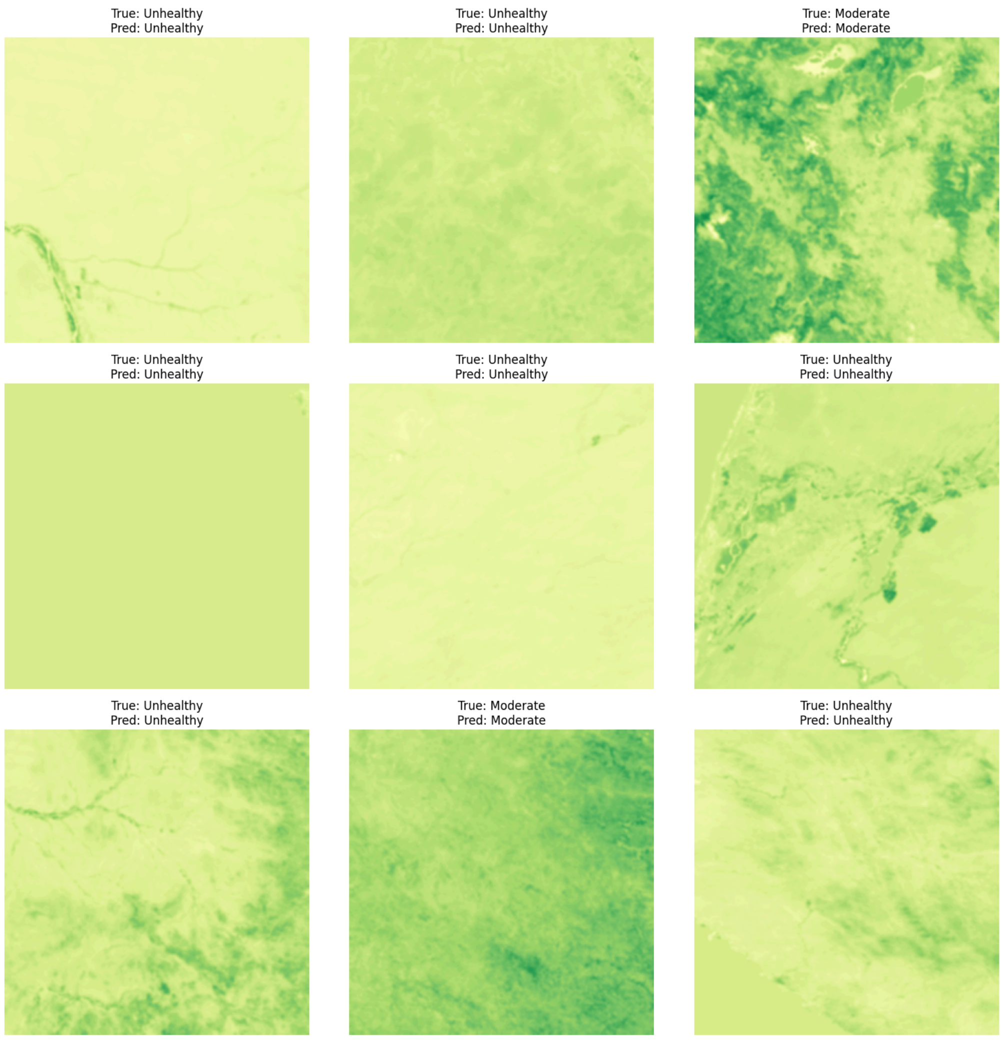
### Training Summary:

The model achieved a training accuracy of **97.83%** with a training loss of 0.0526. On the validation set, it reached a validation accuracy of **92.99%** and a validation loss of 0.0526. The learning rate during this stage was 1e-5.

## Multi-Format Preprocessing

### Supported Input Types:

- **TIFF/TIF**
- **NPY**
- **RGB Images: Convert to VARI (Visible Atmospheric Resistance Index)**

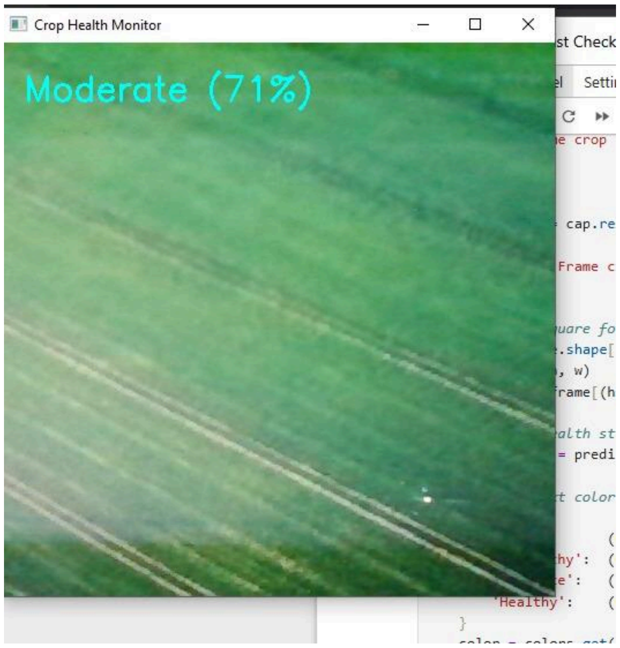

True: Unhealthy
Pred: Unhealthy
True: Unhealthy
Pred: Unhealthy
True: Moderate
Pred: Moderate
True: Unhealthy
Pred: Unhealthy
True: Unhealthy
Pred: Unhealthy
True: Unhealthy
Pred: Unhealthy
True: Unhealthy
Pred: Unhealthy
True: Moderate
Pred: Moderate
True: Unhealthy
Pred: Unhealthy

## Real-time crop health monitoring

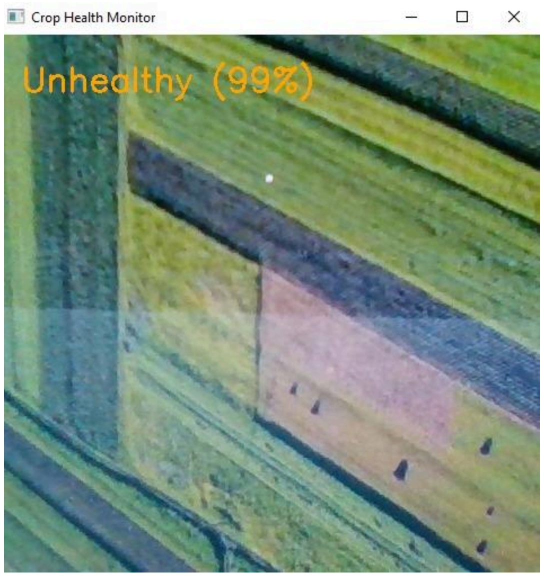
**Real-Time Crop Health Monitoring Module**

**This component captures live video from the laptop's webcam, computes an NDVI-like vegetation index on each frame, and classifies crop health into four categories—Dead, Unhealthy, Moderate, and Healthy—using a pretrained Inception-based CNN. The predicted label and confidence percentage are overlaid on the live video feed.**

## Result:



```
e crop
 cap.re
Frame c
uare fo
.shape[
, w)
rame[(h
alth st
= predi
t color
                  (
hy':     (
e':      (
'Healthy':     (
}
color = colors.get(
```



## Backend Overview

The backend of the Crop Health Analysis project is a FastAPI-based application designed to provide machine learning-driven crop health predictions and user management functionalities. It serves as the core processing unit for the application, handling data processing, model inference, and API interactions with the frontend.

### Backend Structure

The backend is organized into modular components to ensure maintainability and scalability. Below is the directory structure:

- main.py: Entry point of the FastAPI application. Initializes the app, sets up middleware, and includes routers for different API endpoints.
- routers/: Contains route handlers for different API modules.
    - auth.py: Manages user authentication (signup, login).
    - image.py: Handles image uploads and retrieval from Supabase Storage.
    - log.py: Handles log routes for requests users make.
    - classification.py: Manages crop health classification using a pre-trained machine learning model.
    - users.py: Handles routes to get all users.
- services/: Contains business logic for each router.
    - auth.py: Implements authentication logic (e.g., password hashing, JWT token generation).
    - image.py: Handles image storage and retrieval operations with Supabase.
    - log.py: Handles logs for requests users make.
    - classification.py: Loads the machine learning model (Inception.keras) and performs crop health predictions.
- model/: Contains logic for model preprocessing and the model itself.
    - Inception.keras: classification model.
    - image_processing.py: Contains utility functions for image preprocessing (e.g., NDVI computation, resizing).
    - model_script.py: Handles the flow of how the model interacts with images.

- database/: Manages database interactions.
    - supabase.py: Initializes and manages connections to Supabase for user data and image storage.
- image_processing.py: Contains utility functions for image preprocessing (e.g., NDVI computation, resizing).
- requirements.txt: Lists Python dependencies (e.g., fastapi, uvicorn, tensorflow, supabase).
- Dockerfile: Defines the container setup for deployment on Railway, including environment variables to suppress TensorFlow warnings.
- Docs: containing files for api-documetation and diagrams folder.

## Backend Features

### User Authentication:

- Signup: Allows users to create an account by providing an email and password. Passwords are hashed before storage in Supabase.
- Login: Authenticates users and returns a JWT token for secure API access.
- Implementation: Uses Supabase for user data storage and JWT for session management.

### Image Management:

- Upload: Users can upload images to Supabase Storage (e.g., images bucket).
- Retrieval: Fetches images from Supabase for processing or display.
- Implementation: Integrates with Supabase Storage for scalable image handling.

### NDVI Calculation:

- Functionality: Computes NDVI values from uploaded images to assess crop health based on vegetation indices.
- Implementation: Uses image_processing.py to preprocess images and calculate NDVI.

### Crop Health Classification:

- Functionality: Uses a pre-trained Inception.keras model to classify crop health (e.g., healthy, diseased) based on image data.
- Implementation: The model is downloaded at runtime from Supabase Storage and loaded using TensorFlow in services/classification.py.

## Backend Functionality

### Data Processing:

- Handles image uploads, NDVI calculations, and crop health predictions.
- Uses Supabase for persistent storage of user data and images.

### Machine Learning:

- Downloads the Inception.keras model from Supabase Storage at startup.
- Performs inference on uploaded images to predict crop health.

### API Serving:

- Exposes RESTful endpoints for the frontend to interact with.
- Ensures secure communication using JWT-based authentication.

# Frontend Overview

The frontend of the Crop Health Analysis project is a Streamlit-based web application that provides a user-friendly interface for interacting with the backend API. It allows users to sign up, log in, upload images, and view crop health analysis results.

## Frontend Structure

The frontend is a single-page application (SPA) built with Streamlit, which simplifies the development of data-driven web apps. The structure is as follows:

- ui.py: Main entry point for the Streamlit application. Defines the UI layout and handles user interactions.
- requirements.txt: Lists Python dependencies for the frontend (e.g., streamlit, requests).

## Key Frontend Features

- **User Interface:**
  - Login/Signup Pages: Forms for user authentication, interfacing with the backend's /auth/signup and /auth/login endpoints.
  - Image Upload: Allows users to upload images for analysis, interfacing with the /image/upload endpoint.
  - Analysis Dashboard: Displays NDVI values and crop health predictions, fetched from the /ndvi/calculate and /classification/predict endpoints.
- **Responsive Design:**
  - Streamlit provides a simple, responsive layout that works across devices.
- **Interactive Elements:**
  - File upload widgets for image uploads.
  - Buttons to trigger NDVI calculations and crop health predictions.
  - Visualizations to display NDVI values and prediction results (e.g., text output, charts if implemented).

## Frontend Functionality

- **User Authentication:**
  - Collects user credentials and sends them to the backend for signup/login.
  - Stores the JWT token in Streamlit's session state for authenticated requests.
- **Image Upload and Analysis:**
  - Provides a file uploader for users to submit images.
  - Sends the image to the backend for storage and processing.
  - Displays the results of NDVI calculations and crop health predictions.
- **API Integration:**
  - Uses the requests library to interact with the backend API endpoints.
  - Handles responses and renders them in the Streamlit UI.

## Integration Between Backend and Frontend

- **API Communication:**
    - The frontend communicates with the backend via RESTful API calls, using the endpoints documented in the API docs.
    - Authentication is managed using JWT tokens, which the frontend includes in the Authorization header for protected endpoints.
- **Data Flow:**
    - User uploads an image via the frontend.
    - The frontend sends the image to the backend's /image/upload endpoint.
    - The backend stores the image in Supabase and returns an image_id.
    - The frontend uses the image_id to request NDVI calculations (/ndvi/calculate) and crop health predictions (/classification/predict).
    - The backend processes the requests and returns results, which the frontend displays to the user.

## Team Members:

**Abdulaziz Abdul Tawab Muhammad**

**Esraa Basher Abdelgawad**

**Eslam Mohamed Said Ali**

**Haya aboulwafa**