

FOLIAGEFIXER

Classification and Segmentation of Tomato Foliar Diseases

Funky

Hari Murti-Baran-816014286

Ojore Kanneh-816026687

Ved Mahadeo-816025586

Final Project

INFO3604

Department: DCIT

Supervisor Dr. Vijayandh

Rajamanickam

Date Submitted 18th April, 2023

TABLE OF CONTENTS

<i>FoliageFixer</i>	1
<i>Abstract</i>	4
<i>Introduction</i>	6
Problem Description	6
Positioning	8
Stakeholder Descriptions.....	8
Product Overview	9
Business Aspects.....	9
<i>Requirements Specification</i>	10
Functional Requirements	10
Non-Functional Requirements.....	10
User Stories.....	10
Sequence Diagram	11
Use Case Diagram	12
Entity Relationship Diagram	12
Technical Constraints	14
<i>Design Specification</i>	14
System Architectural Design.....	14

User Interface Design	25
<i>Implementation</i>	30
<i>Test</i>	49
<i>Conclusions, Lessons Learned and Recommendations</i>	52
<i>Git repo and Trello</i>	54
<i>References</i>	56
<i>Appendix</i>	<i>Error! Bookmark not defined.</i>

ABSTRACT

Crops affected by foliar disease can severely impact crop quality and reduce a farmer's output and hence profit. These farmers depend on the profitability of their produce. It is crucial to keep track of plant quality, but it can be time consuming and requires some experience and expertise to be able to effectively recognize/classify diseases and come up with a management strategy.

This is also subject to human errors. Additionally, researchers often study these plants.

Classification is not as difficult in this case. However it is sometimes necessary to calculate the severity of the disease, i.e. the percentage of the leaf surface affected by the identified disease.

This task is not trivial and can also be time consuming.

Thus, we propose our FoliageFixer application to accomplish both tasks. Given the 12-week time constraint, we have limited our domain to tomato plants and for only 7 diseases.

Previous research uses a variety of techniques. Beron, Salcedo, and Martínez (2020) uses a color thresholding to segment and k-means to classify. While the thresholding method is appropriate for the plantvillage dataset which consists of images of leaves against plain backgrounds, these methods tend to fail when given more complex backgrounds which are common to images of leaves taken in the field. Thus, Gonçalves et al. (2021) proposes a deep learning, convolutional neural network (CNN) semantic segmentation approach to disease segmentation.

Our approach to the FoliageFixer combines these two methods. Our semantic segmentation CNN model is trained on a custom labelled dataset and the classification CNN model is trained on the output of the segmentation model.

Our final system comprised of an Android app, a flask API and segmentation model of 85% precision in predicting the diseased class and a classification model of 73% precision in predicting disease.

INTRODUCTION

Problem Description

- Problem Description (5 marks)

Describe the problem or need that the team has addressed. Identify the purpose/objective of the project, the context and the general technical problem the team was solving.

Describe tasks performed. Include information about any research analysis undertaken.

Identify resources required, major risks, **task schedule** and major milestones.

Crops affected by foliar disease can severely impact crop quality and reduce a farmer's output and hence profit. These farmers depend on the profitability of their produce. It is crucial to keep track of plant quality, but it can be time consuming and requires some experience and expertise to be able to effectively recognize/classify diseases and come up with a management strategy. This is also subject to human errors.

Additionally, researchers often study these plants. Classification is not as difficult in this case, however it is sometimes necessary to calculate the severity of the disease, i.e. the percentage of the leaf surface affected by the identified disease. This task is not trivial and can also be time consuming.

Thus, the objective of this project is to produce an app that quickly and easily detects the disease affecting a user's plant and calculate its severity. Given the 12 week time constraint and the availability during the dry season we have chosen to limit our scope to only tomato plants and 7 diseases: Bacterial Spot, Early Blight, Late Blight, Leaf Mold, Tomato Mosaic Virus, Tomato Yellow Leaf Curl and Septoria Leaf Spot. Additionally, we chose to develop only an Android application due to Hari's previous experience with Android development.

Specifically, we attempted to solve the technical problem of high accuracy foliar disease image classification and segmentation using neural networks.

To accomplish this, we had to complete numerous tasks. We began with a research phase to ensure our project was feasible for the time constraint and our ability, as well as formulate our methodology. Then, we began our system design which included the generation of system design diagrams, wireframes and API spec. Once these diagrams were agreed upon, we selected our technologies and began implementation.

The resources required included:

1. laptops for development purposes,
2. at least one Android device for testing purposes,
3. High performance GPUs for model training
4. Dataset of tomato leaf images

The major milestones included:

1. Selection of classification method
2. Selection of segmentation model
3. Training of segmentation model
4. Training of classification model
5. Completed classification sprint – includes image capture on device, image received on API then the image is classified
6. Finalized API
7. Finalized App

Positioning

Our research analysis step included studying the existing market. Plantvillage Nuru among a few other apps have an AI equipped for identifying a limited set of foliar diseases such as cassava and potato. However, none of these apps perform the segmentation and severity calculation. We believe this is our competitive edge.

Stakeholder Descriptions

Our stakeholders include amateur and professional farmers and researchers in the life sciences department.

Amateur tomato farmers have little to no expertise and thus cannot effectively identify the disease affecting their plants. They therefore stand to benefit from the FoliageFixer.

For professional tomato farmers, the quality and quantity of their crop for the season is of great importance, as it directly affects their ability to support themselves and their family. Disease identification for these users is not as difficult, but there is room for a lot of human error.

Researchers such as those in the life sciences department are required to calculate disease severity but this process is tedious for large numbers of leaves. The FoliageFixer aims to alleviate this effort.

Product Overview

The FoliageFixer is an Android application that interacts with a Flask API backend and classification and segmentation neural networks. Its core feature is its ability to detect the disease affecting a tomato leaf, calculate its severity and provide disease management strategies. It also provides additional convenience by providing a recent scans page that returns all recently scanned leaves together with the classification and segmentation results and disease management strategy.

Business Aspects

For this project, we had a budget of zero dollars. This affected our choice of hosting solutions. Additionally, the application is not expected to have any IP or patent issues since it is an Academic project. Currently there is no specific market or industry outlook to consider since the scope of the project is focused on solving a specific problem.

REQUIREMENTS SPECIFICATION

Functional Requirements

- Users will be able to take pictures of leaves.
- The system shall analyze the leaf to detect and classify if it is diseased.
- The system shall present the user with the findings of the analysis along with any disease management practices if necessary.
- If the leaf is diseased the system shall compute the severity of the disease as a percentage.

Non-Functional Requirements

- Accuracy - at least 95% accuracy on validation data
- System must not require users to have the highest quality cameras.

User Stories

1. A farmer notices some yellow spots on his tomato plants but does not know what it is.

The next day, the spots grow in size and he begins to worry. He asks some other farmers for their advice but they can't agree on what could be causing it or what treatment to apply. He decides to use the FoliageFixer app. On opening the app he is immediately presented with the camera interface. He uses this to take a photo of one of the diseased leaves. After taking the photo the app displays a screen showing the exact name of the disease, the severity of the disease as a percentage and a treatment plan.

2. A researcher in the life sciences department at UWI specializes in foliar diseases. After a day on a field she collects 50 tomato leaf samples both diseased and healthy. As an expert in the field she can manually diagnose the diseases if she chooses, however it will be time consuming. However, she also wants to know the % severity. So, she opens the app and

scans each leaf to get the disease diagnosis and % severity. She also has the added benefit of being able to view her scan history in the app, making it easy for her to record the data.

Sequence Diagram

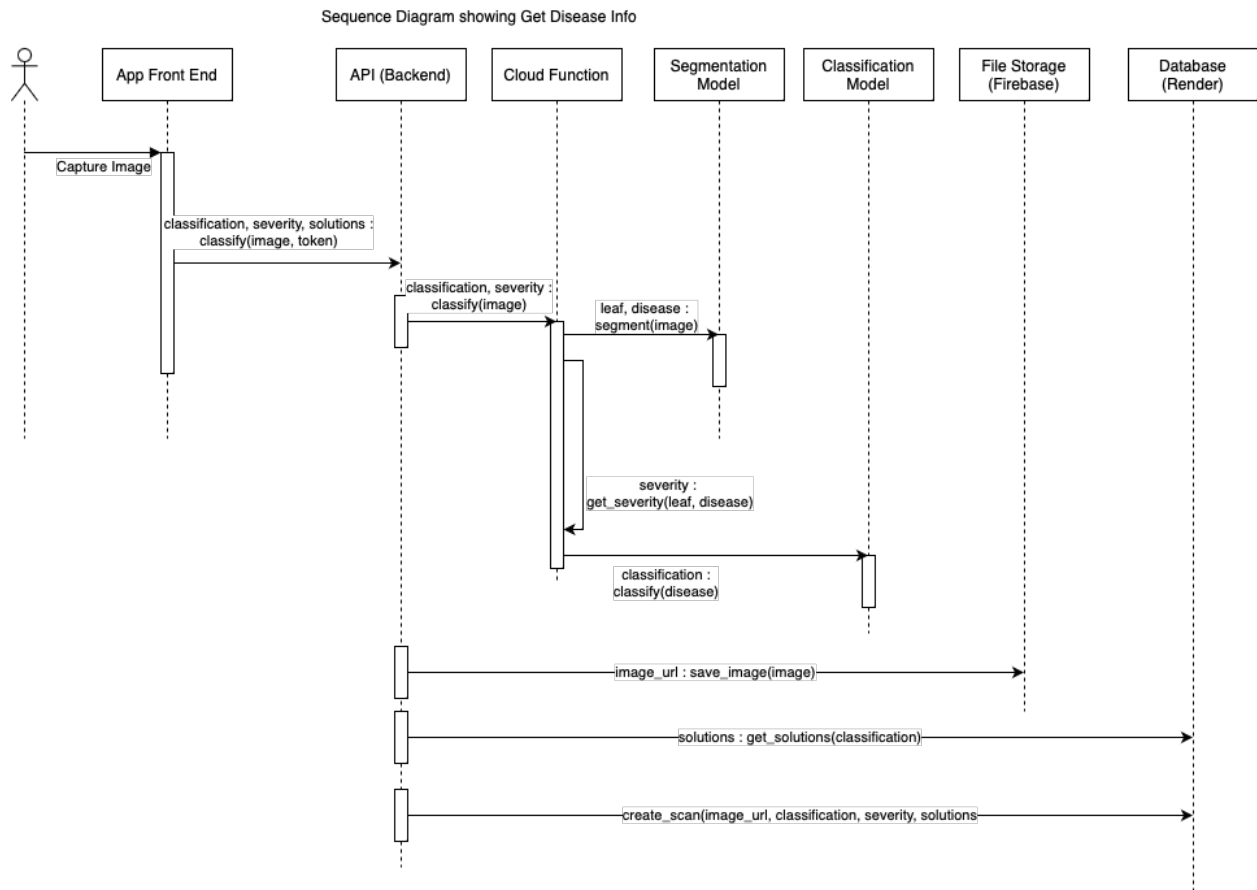


Figure 1: Sequence Diagram

Use Case Diagram

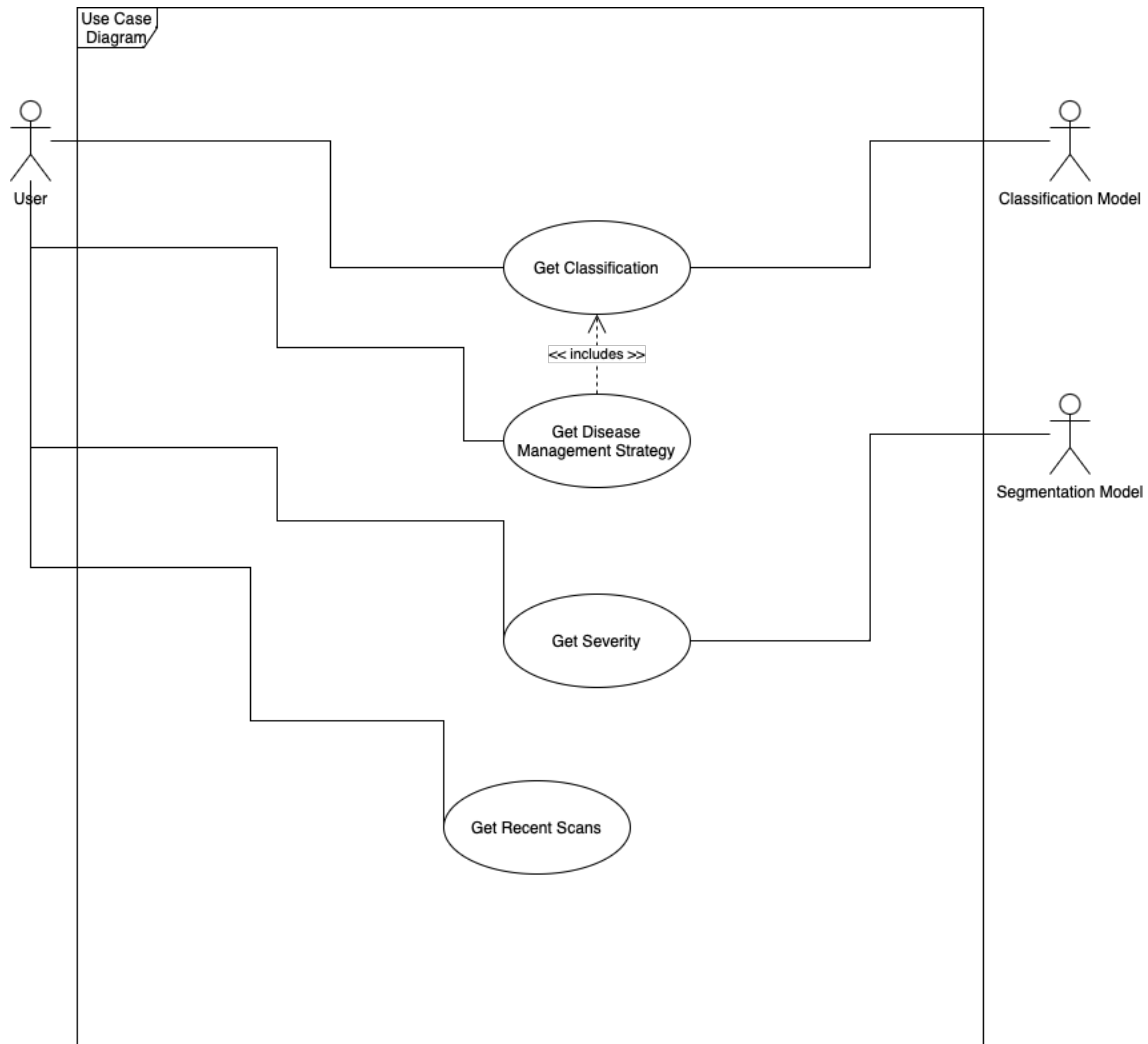
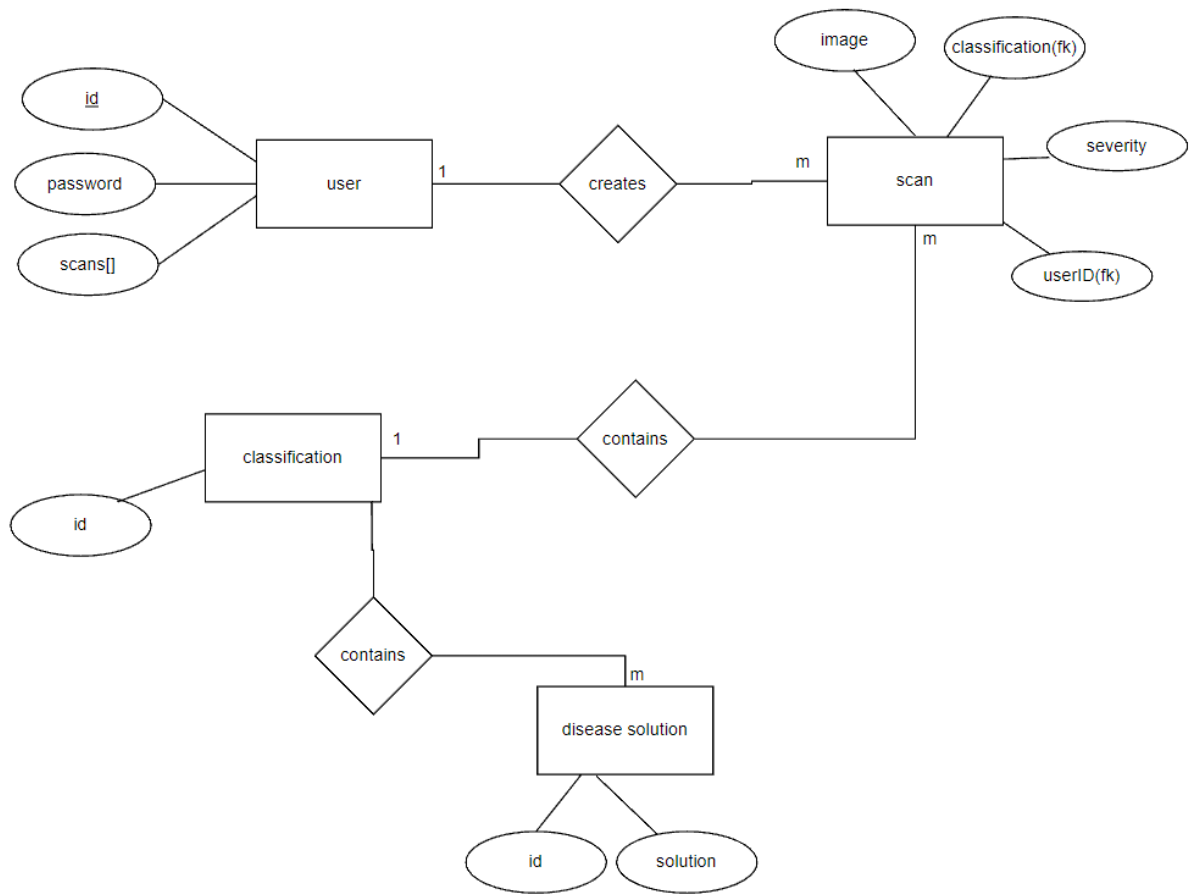


Figure 2: Use Case Diagram

Entity Relationship Diagram



Technical Constraints

Our main technical constraints arise from the very small budget we are working with. For one, our deployed models don't use a GPU as it is difficult to get GPU resources for free. This hinders the inference speed. Additionally, free plans for databases and file storage have harsh usage limits. Firebase only allows 1GB file storage and Render only allows 1GB database storage and it is automatically reset every 90 days.

DESIGN SPECIFICATION

System Architectural Design

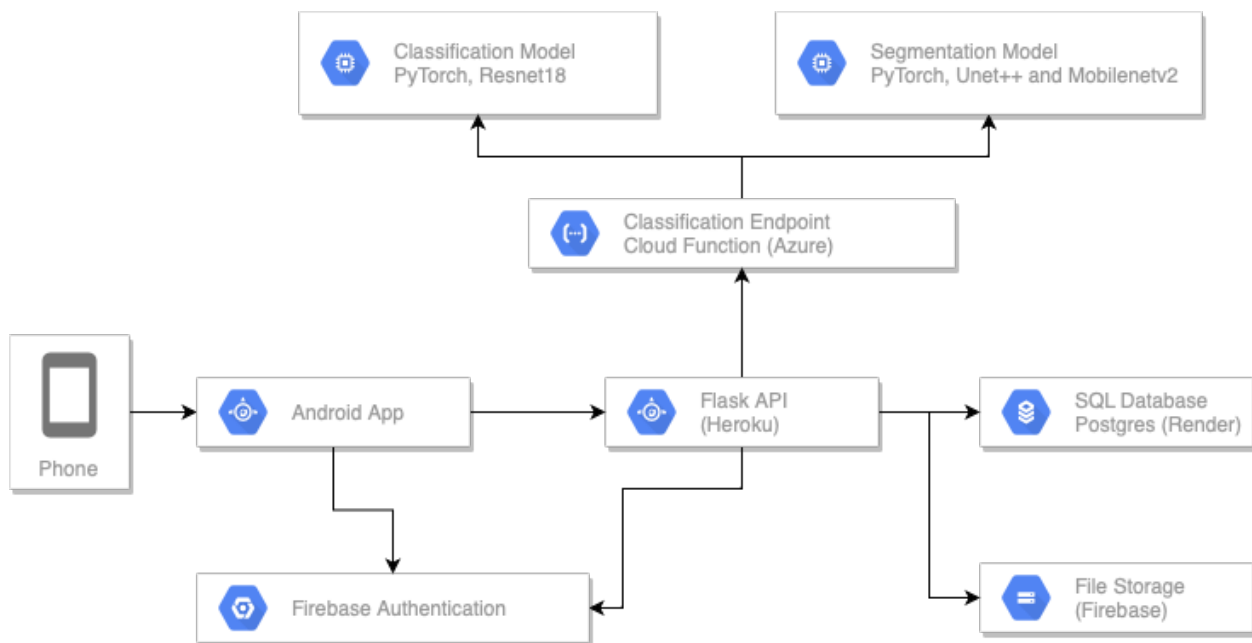


Figure 3: System Design

Alternative designs may include deploying the classification and segmentation models together with the API. However, due to budget constraints this was an issue. The free Heroku dyno does

not allow a slug size over 500MB. Additionally, it only has 1GB of ram which is significantly less than required by the models.

Additionally, a NoSQL database was also considered due to Firebase's noSQL offering.

However, the relational approach was chosen as we have most of our experience using SQLAlchemy ORM and relational databases such as PostgreSQL.

Back End API

The API Specification

The API spec is a detailed document that outlines how an application programming interface (API) should behave. It typically includes information about the API's endpoints, parameters, request and response formats, and error handling. In our project we made use of several routes that acted as a service when users made requests to get a response on any feature of our application.

User Authentication

1) Register a new user

URL: / adduser

- Method: **POST**
- Request body:
 - **username** (string, required): the username of the new user
 - **password** (string, required): the password of the new user
- Response:
 - Status code 201 Created if successful
 - Status code 409 Conflict if the username is already taken

- This route responds to a request from the front end when a new user performs the signup procedure. When they sign up in the front end using the firebase architecture, a token will be generated which will then be sent to this route requesting for the user of that token to be added to the database. Therefore, this route will receive the token, perform a token decode, to receive the user id of that user and then proceed to add that user to the system database. The response will just be an indication of success or failure to add the new user

2) Log in

URL: **/login**

- Method: **POST**
- Request body:
 - **username** (string, required): the username of the user
 - **password** (string, required): the password of the user
- Response:
 - Status code 200 OK if successful
 - Status code 401 Unauthorized if the credentials are invalid
- This route responds to a request sent by the front end to ensure that the information entered by a user to login is correct or exists. When a user login in the application, they will enter their email and password. This email and password will be received in

the request body of the back end and firebase authentication to ensure that the information sent is present in the firebase user authentication information will be done. When it passes the authentication check, a response with the user token is given back to the front end to show the user is successfully logged in and can perform operations of the application with their respective tokens. When the authentication fails, an error message is returned, and no token is sent giving the user no access to the application.

Image Classification

Send an image for classification

3) URL: **/classify**

- Method: **POST**
- Request body:
 - **image** (file, required): the image file to be classified
 - **User_id**: unique user ID, so the scan can be saved
- Response:
 - Status code 200 OK if successful
 - Response body:
 - **disease** (string): the name of the disease
 - **severity** (float): the severity percentage of the disease
 - **management_strategy** (string): the recommended management strategy

- This route is responsible for returning the actual classification results such as leaf classification, severity and disease management strategy to the user. When the user is using the application and they select an image to be classified and they then click the actual classify button on the application. A request is sent to this route with the user token and the image to be classified. When the user is authenticated through token authentication, a request to the azure cloud function is sent with the image and the classification results is returned along with the severity. The model is stored in that azure function. After receiving the result, the get solution method is called to get the solutions for the specific disease and this information is returned to the front end. A scan object is also created in this route which consists of the classification information, severity, user id and scan id.

Get recent scans

4) URL: **/recent**

- Method: **GET**
- Response:
 - Status code 200 OK if successful
 - Response body:
 - **scans** (array of objects): the list of recent scans, each with the following properties:
 - **id** (integer): the unique ID of the scan
 - **user_id** (integer): the ID of the user who sent the scan
 - **disease** (string): the name of the disease
 - **severity** (float): the severity percentage of the disease

- **solutions** (string): a ‘;’ separated list of solutions
 - **image_url** (string): the URL of the classified image
- This route is responsible for returning all the scans that a specific authenticated user took. When the user selects recent scans in the application, a request is sent with the user token. This token is used to find the user by id. When the user is found, it will call a function to the database to return all the scan objects created under that user model. Not just the contents of the scan object is returned but also the disease management solution is returned. If user token doesn’t pass the authentication check, an error message is sent back to the user.

Cloud Function

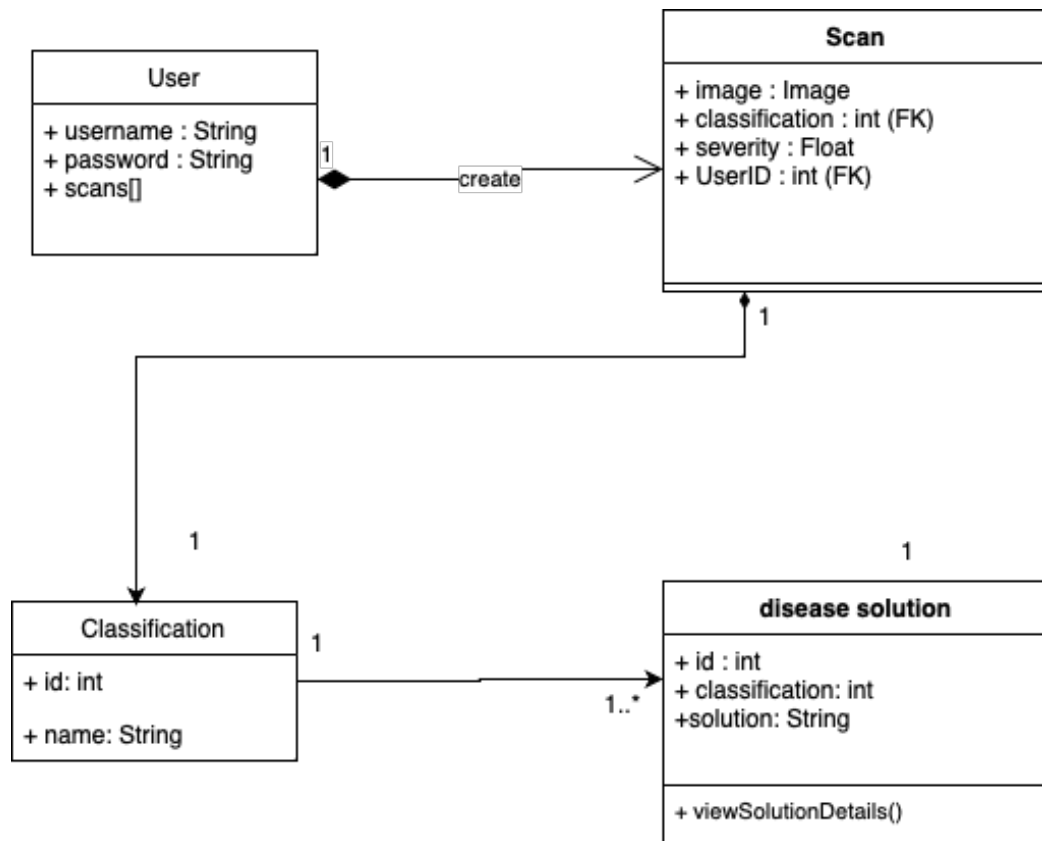
5) URL: <https://foliagefixermodel.azurewebsites.net/api/foliagefixermodel/>

- Method **POST**
- Request body:
 - **image** (file, required): the image file to be classified
- Response:
 - Status code 200 OK if successful
 - Response body:
 - **disease** (string): the name of the disease
 - **severity** (float): the severity percentage of the disease
- This route is responsible for receiving the classification request from the classify route. The azure cloud function houses the actual model and is there the actual classification is

calculated. It will receive an image in the request body and when it calculates the actual classification and severity then will return the results back to the classify route.

Models

Figure 4: Class Diagram



Class diagram showing the layout and structure of the classes used in the application.

Explanation of design

Each instance of User has a username, password and scans attribute. This scans attribute is important as every time a user performs a 'classify' on an image, a new scan object is created in the scan model for the specific user and the scan list increments under the user model. The Scan class holds a list of scan objects containing all the attributes of a scan object which includes the image, classification, severity and user ID which acts as a foreign key to link the user and scan

table together. It was decided that a data table containing only the classification names and their IDs would be necessary so that if we need to add a new classification it can just be added in that table and it will be changed in all the other tables as all the other tables link to this table to get the classification IDs. The solutions is linked to a classification via the classification ID which is a foreign key in this table. A simple way to view this structure of the model is that a user when using the application will create a new scan object when they classify a leaf image, and this new object will contain the classification and severity information.

Classification and Segmentation Models

The system is comprised of two neural network models: segmentation model, classification model. The segmentation model is required to segment an image into healthy leaf, diseased leaf and background which forms the basis of our severity calculation. The classification model then receives the diseased leaf segments and predicts the classification. The diagram below shows this pipeline in more detail.

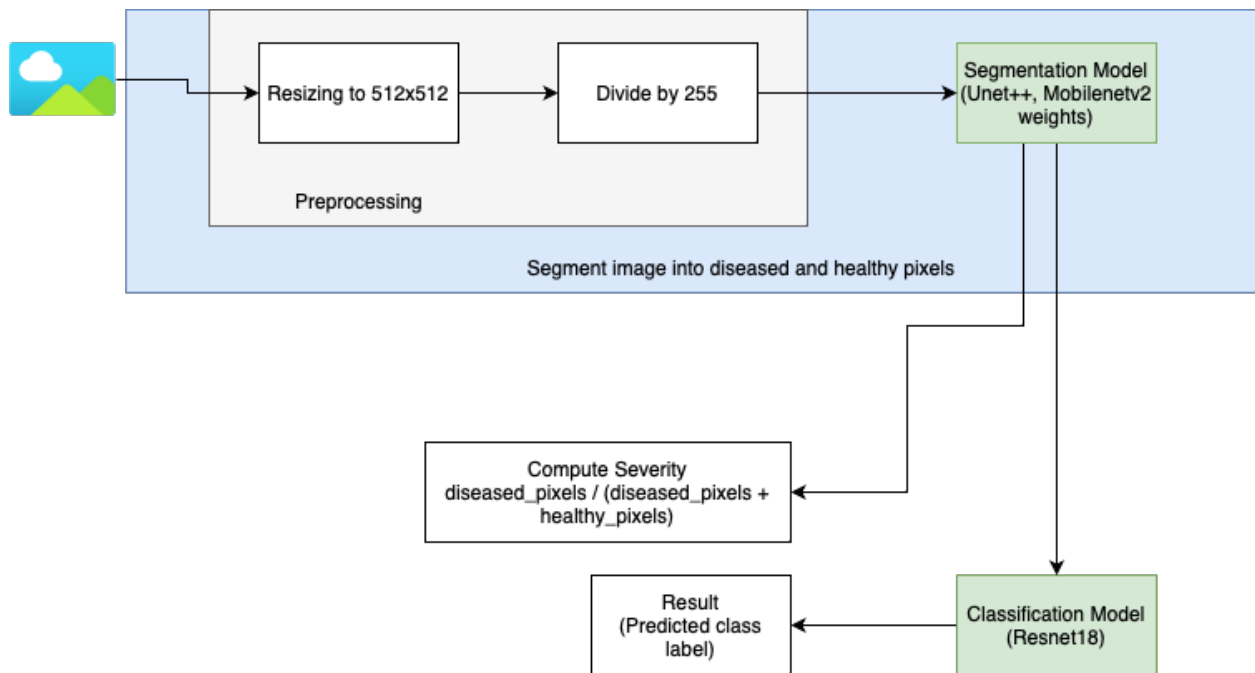


Figure 5: Input Pipeline for Inference

The image first undergoes a resizing step. It is necessary to reduce image size to increase model speed. However, in images with low disease severity, the diseased portion is miniscule and very hard to segment. Thus, we have chosen 512 pixels to retain some of the fine-grained detail.

Classification Model

The classification model makes use of the resnet18 architecture which is an 18 layer residual network which has seen great success in image classification (He et al. 2015). The resnet50

model has also been used in foliar disease classification for sunflowers (Dawod and Dobre 2022). However, for our study we have chosen resnet18 due to the memory and computation limits of Google Colab which was used to create Jupyter notebooks for training the model.

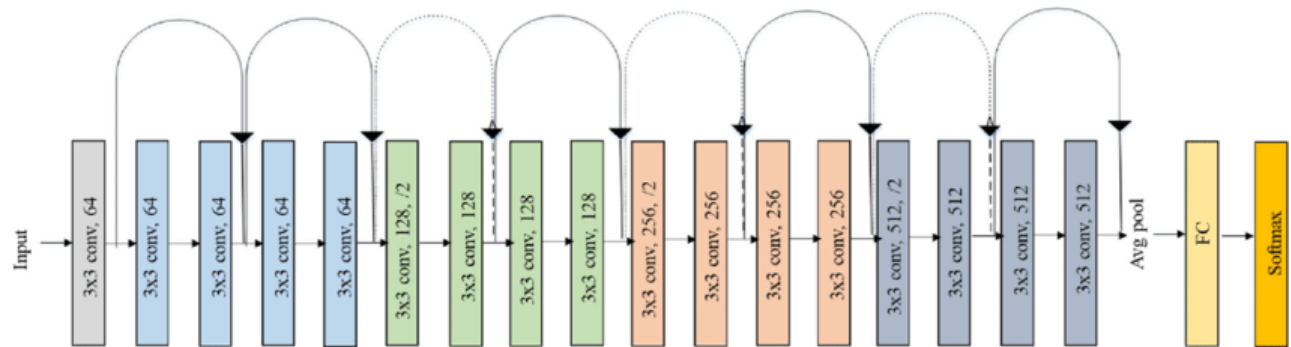


Figure 6: Resnet18

The diagram above shows the architecture diagram of resnet18 (Ramzan et al. 2019).

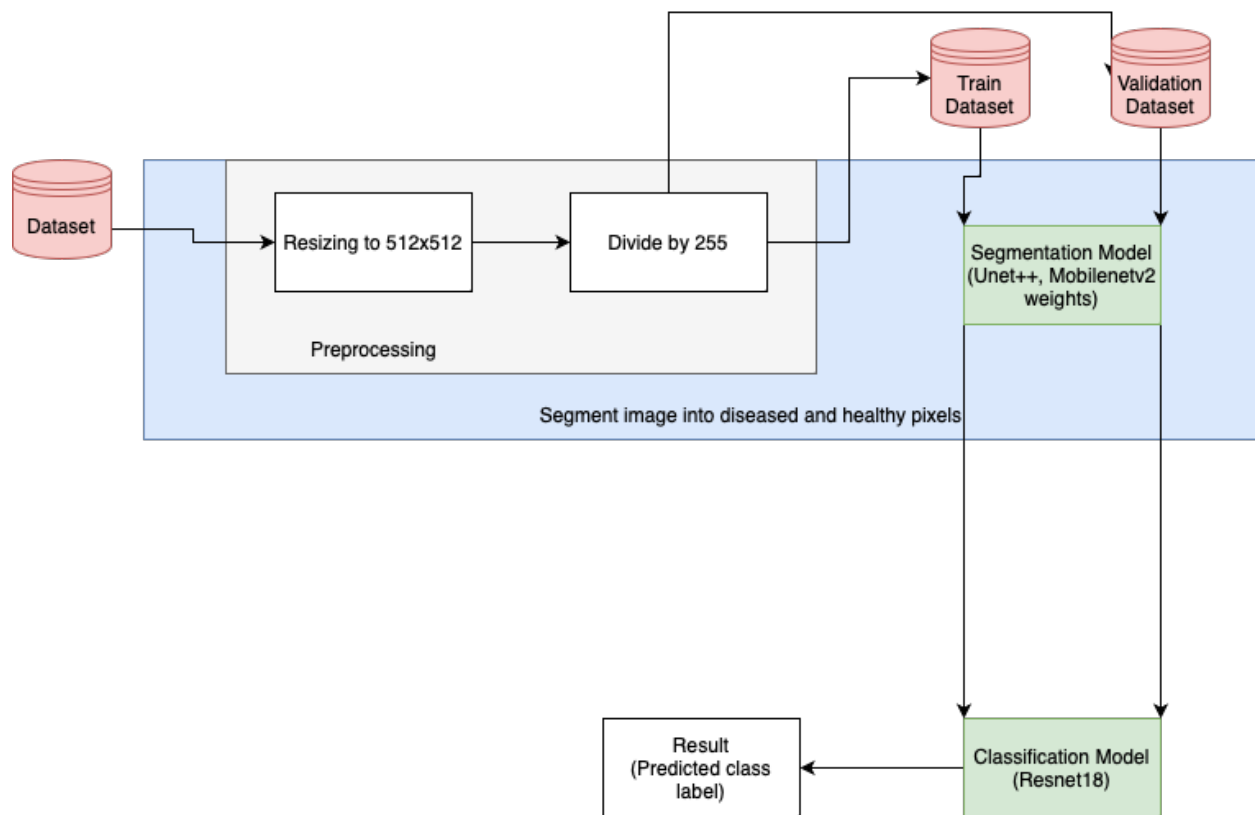


Figure 7: Input Pipeline for Training

The diagram above shows the input pipeline used for training the classification model.

Segmentation Model

While conducting research to decide on the processing pipeline for segmentation we experimented with many methods. Initial attempts included unsupervised methods such as k-means clustering, mean shift clustering and superpixel segmentation. With these methods we were able to achieve decent results on images with simple backgrounds, but poor results on complex backgrounds such as the field images. To solve this, Gonçalves et al. (2021) used a semantic segmentation supervised approach by labelling every pixel of each image, then training a model to predict a label for each pixel in new images. With this approach they achieved excellent results with a 85% precision on the diseased class. This forms the basis of our methodology for segmentation. Gonçalves et al. (2021) compares segmentation results for many different architectures, including Unet. We chose an updated version of the Unet architecture, Unet++ which is purported to improve segmentation (Zhou et al. 2018). There are no publicly available datasets with labelled tomato leaves, so we were required to label our own dataset. Given the time constraints, it would be impossible to label the entire dataset. Therefore, we used transfer learning by training a Unet++ model initialized with weights after the model is trained on the imagenet dataset. Transfer learning improves training when the dataset is small. Unet++ was chosen due to its success in medical segmentation which has a similar issue in having to segment very fine boundaries.

Severity Calculation

The severity is calculated by a pixel counting method following the formula given below.

$$Severity = \frac{disease_pixels}{disease_pixels + leaf_pixels}$$

User Interface Design

The User Interface of the application has been designed to be intuitive and user friendly. It features a modern and simple design that is visually appealing and easy to navigate. A consistent color scheme is maintained throughout the application with a dark forest green (#00352A) as the primary color for buttons, toolbars, and card backgrounds along with white text to contrast the dark forest green of application objects. The application also features a neutral grey, off-white mix for the page backgrounds (#F7ECE9) to reduce eyestrain.

The application is split into three main sections:


- The Home Screen: displays three cards each representing a different action the user can take.
- The Gallery screen displays a grid of all the user's application related photos (application related photos- photos captured in the application or imported for processing). When a gallery image is selected, the user will be taken to the image processing screen where the image along with the processing results will be displayed.
- The Recent Scans screen: displays a scrollable list of the 10 most recently processed images. When a scan list item is clicked, the user will be taken to a screen that displays an expanded image along with processing results.

Additionally, the primary form of navigation apart from default android navigation bar and gestures is the application Title which is present in the toolbar on all screens. When the title is pressed, it will carry the user back to the application's home screen.

Wireframes

Login

foliagefixer




sign in

Login

[Forgot Password?](#) [Create Account](#)

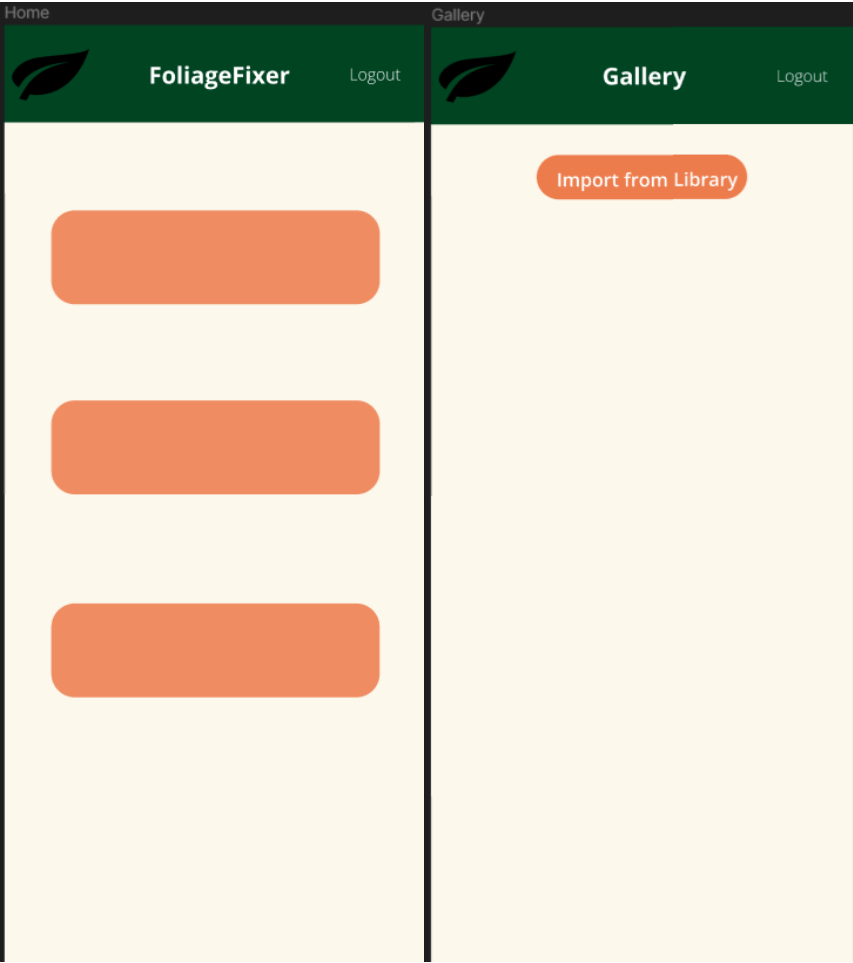
Signup

foliagefixer



sign in

Create





Objects and Actions

The objects included below was planned when designing the app layout with wireframes:

Sign in Screen:

- Create Account text – takes user to the create Account screen.
- Login Button – authorizes entered user credentials and if successful carries the user to the application's home screen.

Create Account:

- Create Account button – creates user with firebase and sends user credentials to flask backend, after which the user is taken to the login screen.

Home Screen:

- Capture Card- when pressed will open the phone's camera and allow the user to capture an image. Captured images are stored locally on device storage, as well as send to the flask backend for processing.
- Recent Scans Card – when pressed will take the user to a list of recently processed images.
- Gallery Card – when pressed will take the user to the application's gallery. (Screenshot 8)

Recent Scans Screen:

- Scan list card – when pressed will take the user to the scan details page for the respective list item

Gallery Screen:

- Import From Gallery Button – loads the phone's gallery application to allow users to select an image to import into the application.
- Image card:
 - When pressed will take the user to the image processing page (screenshots 4,5).
 - When long pressed will delete the image.

Toolbar:

- Application title when pressed will take the user back to the home screen
- Logout text when press will sign the user out of the application and take them back to the sign in screen.

IMPLEMENTATION

Back End API

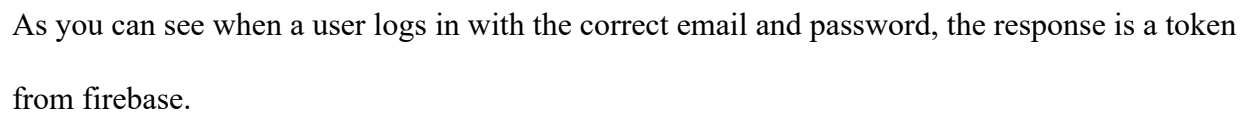
For the Back End API, Python coding language was used with the flask framework. In this framework, we used the MVC template which stands for model, view and controller. With this template, the back end was split into three parts. To begin with, in models, it consisted of all the database tables using SQL Alchemy to interact with the database. The controllers consisted of all the method functions to perform the database queries which consisted of all the commits to the database and all the select*from – where queries. In the views, all the route files were located which were responsible for the request and responses for the front end of the application. Since we used firebase authentication and image storing in our application as well, we had to use certain libraires to import firebase into the API.

The libraries:

- Flask library- responsible for all the functions flask has to offer
- Firebase_admin- used for all the firebase authentication and image storage.

In implementation of the API routes, we used the Postman software to make sure the routes were taking in the responses correctly and performing the correct operations and giving the expected result.

/login



/classify

The screenshot shows a REST client interface for a project named 'ojo-dev'. The active tab is 'POST /classify'. The request is a POST to '({host})/classify' with form data. The form data includes an 'image' field with a file named 'e4Fu1aF-A/pigeon.jpg' and a 'user_id' field with the value 'oG5SMRofOQy91KUpvUWvUWKDI5b2'. The response is a JSON object with the following structure:

```
1 {
2   "classification": "Tomato Mosaic Virus",
3   "classification_id": 7,
4   "image_url": "https://firebasestorage.googleapis.com/v0/b/foilagefixer.appspot.com/o/images%2Ffc0713091edf41178522903a025f8d2f?alt=media&token=5cbc755c-682b-b2da-3305364c447d",
5   "severity": 25.78292752896585,
6   "solutions": "You can try covering your plants with a floating row cover or aluminum foil mulches to prevent these insects from infecting your plants;soak seeds of susceptible plants in a
7     10%% bleach solution before planting."
8 }
```

The status of the request is 200 OK, with a time of 3.67 s and a size of 677 B.

As you can see the classify route returns the classification, severity and solution information when the route receives and image along with the token which is sent in the header section

/recent

The screenshot shows a REST client interface with the following details:

- URL:** `{{(host)}}/recent`
- Method:** GET
- Headers:** 7 headers are listed, with 'authorization' set to `{{(token)}}`.
- Body:** The response is a JSON array of two objects, each representing a scan. The first object has an id of 7, and the second has an id of 10. Both objects include classification, image, severity, solution, and user_id fields.
- Status:** 200 OK, Time: 519 ms, Size: 1.24 KB

```
1 {
2   {
3     "classification": {
4       "classification": "Tomato Mosaic Virus",
5       "id": 7
6     },
7     "id": 9,
8     "image": "https://firebasestorage.googleapis.com/v0/b/foilagefixer.appspot.com/o/images%2Ffc8713891edf41178522983a025f8d2f?alt=media&token=5cbc755c-882b-44f6-b2da-3385364c447d",
9     "severity": 25.78292752896585,
10    "solution": "You can try covering your plants with a floating row cover or aluminum foil mulches to prevent these insects from infecting your plants;soak seeds of susceptible plants in a 10%% bleach solution before planting.",
11    "user_id": "JUpRahiwGyXn50gK65ZwqNE4TuW2"
12  },
13  {
14    "classification": {
15      "classification": "Tomato Mosaic Virus",
16      "id": 7
17    },
18    "id": 10,
19    "image": "https://firebasestorage.googleapis.com/v0/b/foilagefixer.appspot.com/o/images%2F988609bda296463e8f4275891c012137?alt=media&token=034fb2f8-b941-456f-82d4-7b6e7bf3af3a",
20    "severity": 25.78292752896585,
21    "solution": "You can try covering your plants with a floating row cover or aluminum foil mulches to prevent these insects from infecting your plants;soak seeds of susceptible plants in a 10%% bleach solution before planting.",
22    "user_id": "JUpRahiwGyXn50gK65ZwqNE4TuW2"
23  }
24 }
```

The recent scans route also work as you can see once the user is authorized with a token, the list of his scans will be returned.

Classification and Segmentation Models

Data

Most research papers on foliar disease classification using computer vision use the PlantVillage dataset. However, this dataset contains images of leaves against plain backgrounds only and models trained on this dataset alone perform poorly when introduced to new images.

Additionally, the dataset appears to have high bias (Noyan 2022). For these reasons, we opted to use a modified version of a different dataset that contains both field images and images with plain backgrounds (“Tomato Disease Multiple Sources | Kaggle” n.d.). It contains 25851 images in the training set and 6684 in the validation set (“Tomato Disease Multiple Sources | Kaggle” n.d.). Each split contains 11 folders. It combines images of tomato leaves from multiple sources including PlantVillage. The dataset used for our model was created by removing the Target Spot, Spider Mite and Powdery Mildew classes due to a lack of images in the field and very little visual indicators. Finally, the dataset contained 21257 train images and 5534 validation images with 8 classes total, 7 diseases.

As semantic segmentation is a supervised learning technique that aims to predict a class label for each pixel in an image, it was necessary to label the pixels of as many images in the dataset as possible. Following similar methodology to Gonçalves et al. (2021), we decided to use 3 classes: Background, Healthy and Diseased. Each class was assigned an integer from 0 to 2 respectively. Our chosen labelling tool was Hasty.ai which is a very powerful data annotation tool that incorporates constantly learning AI models to speed up the annotation process (Hasty 2023). Using Hasty we were able to label 126 images, forming our dataset for segmentation model training. The dataset was exported from Hasty in the COCO format which is structured as

follows:

The json files, coco-train.json and coco-val.json are json documents that specify the integer labels for each pixel in each image in the train and validation datasets respectively. The images folder contains all the original images.

Training

Dataset/

images/

train/

val/

annotations/

coco-train.json

coco-val.json

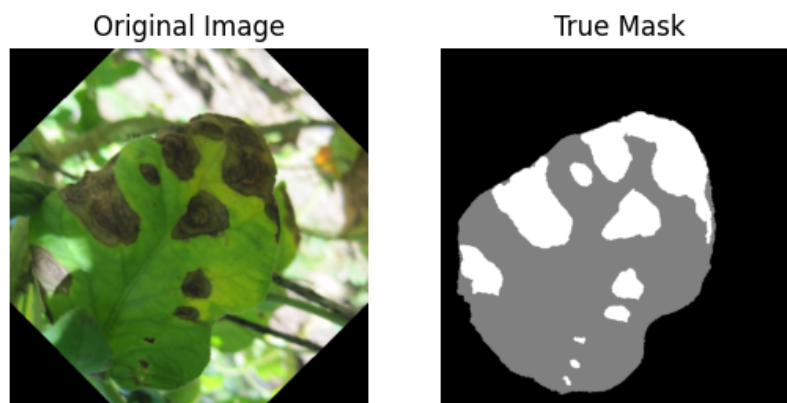
Python was used due to its popularity in machine learning and the extensive selection of libraries. All code used to train the models was written in Jupyter Notebooks. PyTorch and Tensorflow are the most popular python libraries for machine learning. While Tensorflow has superior visualization tools, we chose PyTorch as we found it more intuitive to use (Paszke et al. 2019). The Unet++ implementation with its pretrained weights were sourced from an open source library “segmentation-models-pytorch” (Lakubovskii 2019) and the resnet18 implementation was sourced from PyTorch’s torchvision library (TorchVision maintainers and contributors 2016). Numpy was also used for matrix and vector manipulation (Harris et al. 2020).

For model training, Jupyter Notebooks were written and executed in either Google Colab or Azure Machine Learning. Google Colab is faster as it uses a Tesla T4 GPU but has daily usage

limits and Azure ML has a less powerful Tesla K80 GPU but no usage limits for the first month of usage.

Segmentation Results

The segmentation model was trained for 60 epochs and with PyTorch's implementation of the Adam optimizer. The chosen loss function was the Jaccard Index Loss as it is known to improve model convergence during training as opposed to pixel-wise loss functions such as cross entropy (Wang and Blaschko 2023). In a single notebook, "FFSegmentation.ipynb" each image and its respective "mask" is loaded into memory using the pycocotools python library. A "mask" refers to an image that contains the class label of each pixel. It is in the exact same shape, size, and orientation of the corresponding unlabeled image. The true mask is the image labelled in Hasty. The predicted mask will be the model's prediction.



Augmentations are applied to the dataset using torchvision's transforms feature. Augmentation is proven to improve model generalization. In this case we used random horizontal and vertical flipping, random sharpness adjustment and random rotation as these are all transformations that can be expected in real world situations. The dataset is loaded into a PyTorch data loader that allows for lazy loading of the data. The batch size is 8 due to memory constraints. Afterwards, the model, optimizer and loss function is initialized. A custom training loop is written to iterate

through the dataset and pass each batch of data through the processing pipeline. The training loop consists of two parts: training and validation. The training step preprocesses the input, passes it through the model, then computes the jaccard loss by passing the images and masks to the loss function. The optimizer adjusts the model parameters' weights. During the validation step, the optimizer does not adjust the weights but the loss is computed to track model performance. Additionally, a learning rate scheduler is included such that the learning rate is decreased when the validation loss plateaus for 3 epochs. This is done because a plateaued validation loss implies that the model is no longer improving, and the optimizer maybe stuck in a local optimum rather than the global optimum.

Epoch	Batch Size	Learning Rate	Loss	Optimizer
60	8	0.001	Jaccard	Adam

Figure 8: Summary of Segmentation Model Hyperparameters

After training is completed, the model's loss is plotted.

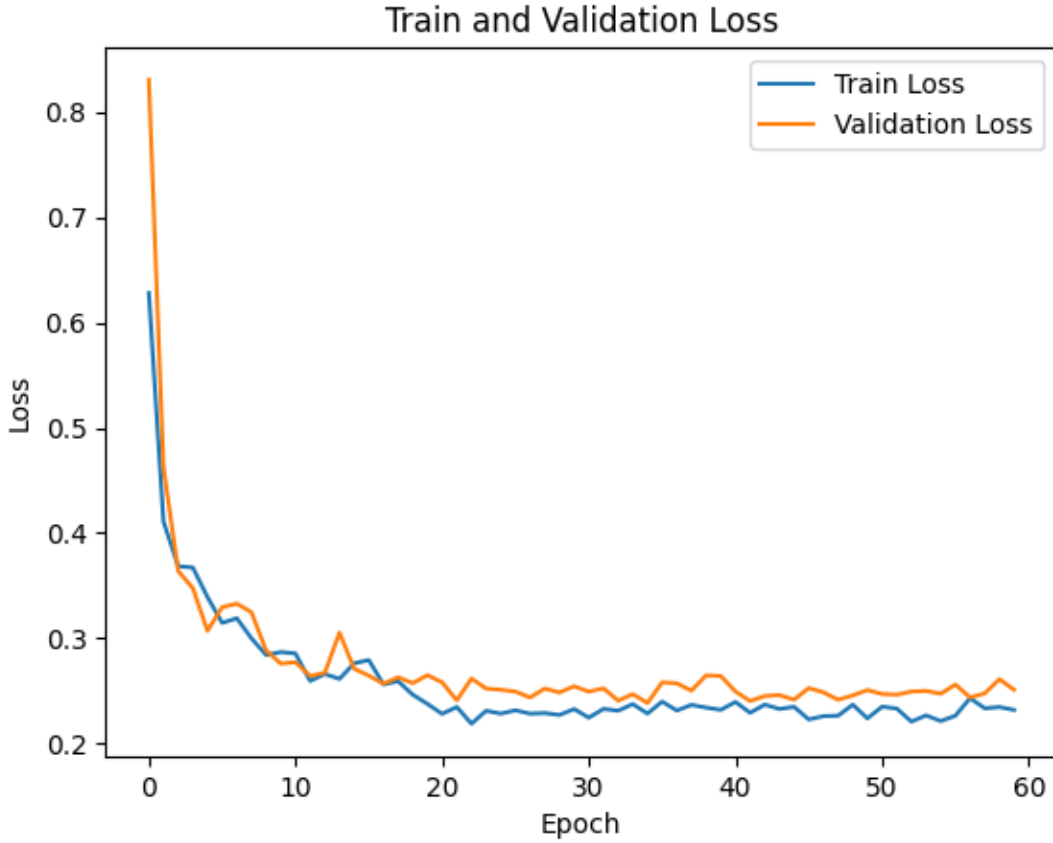


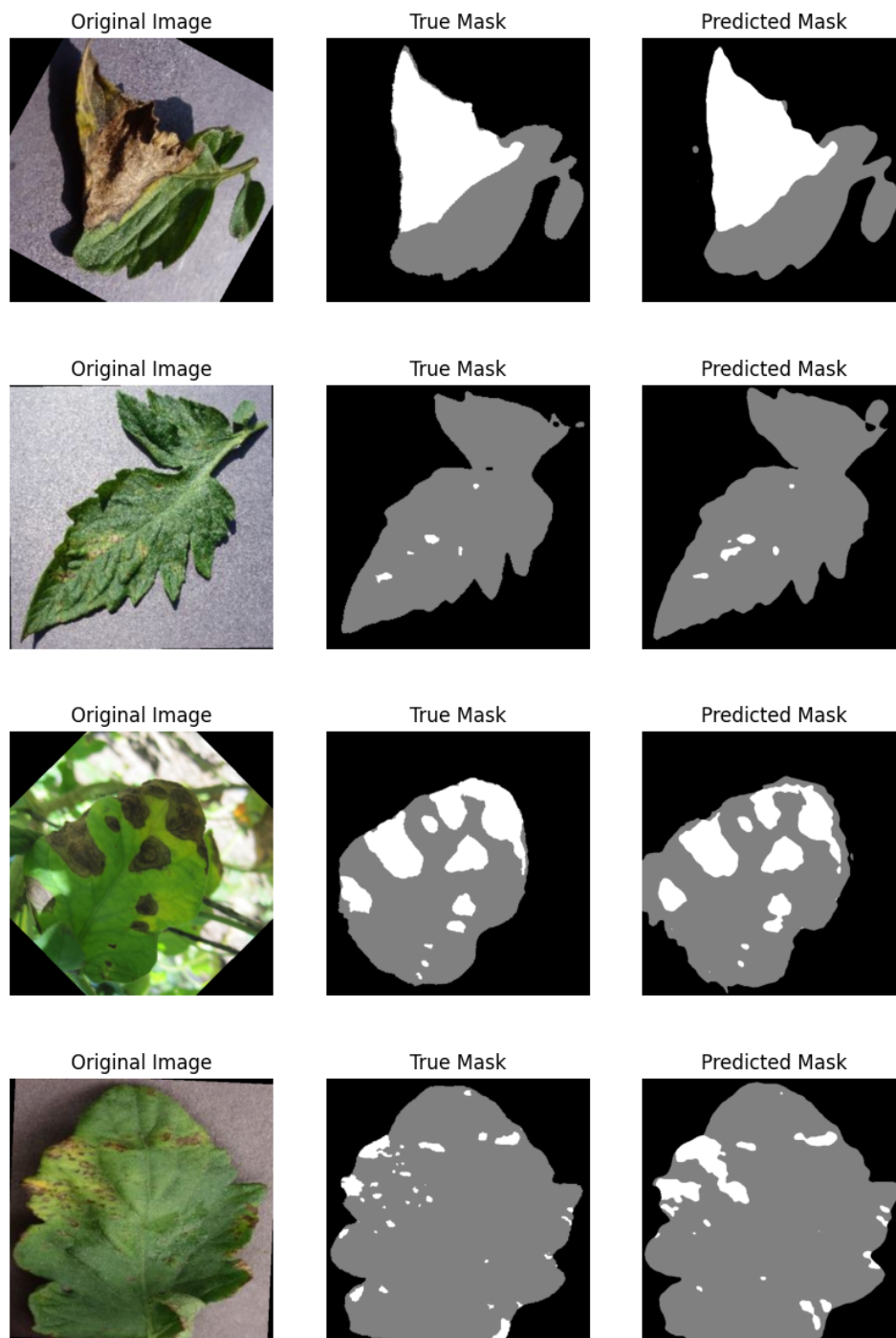
Figure 9: Segmentation Model Loss Plot

The model's precision, recall and accuracy were finally computed as part of an evaluation step. The results for each class for our trained model compared to the Unet used in Gonçalves et al. (2021) is shown below.

Class	FoliageFixer Unet++			(Gonçalves et al. 2021) Unet		
	Precision	Recall	Accuracy	Precision	Recall	Accuracy
0	0.9757	0.9599	0.9629	0.994	0.993	Not Calculated
1	0.8647	0.9510	0.9291	0.971	0.961	Not Calculated
2	0.8256	0.4770	0.9610	0.892	0.859	Not Calculated
Mean	0.8887	0.7960	0.9510	0.952	0.9380	

Figure 10: Results of trained segmentation model

Notably, the Goncalves Unet performs slightly better in each metric. However, this is suspected to be an issue due to our small dataset. Gonçalves et al. (2021) uses a train dataset of 612 labelled images and 154 validation images, as opposed to our very small 103 training and 23 validation. The images below show some sample true and predicted masks.





Classification Results

The classification resnet18 model was trained for 15 epochs, with a learning rate of 0.001, the Adam optimizer and a cross entropy loss function. The code for this model is in the “segClassModel.ipynb” notebook. First, the train and validation dataset is loaded into memory by using pytorch’s dataset ImageFolder utility. These dataset objects are then used to create the train and validation dataloaders. The segmentation model is loaded into memory and the classification model initialized. The training loop follows the same structure as the one outlined in the segmentation results section.

Epoch	Batch Size	Learning Rate	Loss	Optimizer
15	16	0.001	Cross Entropy	Adam

Figure 11: Summary of Classification Model Hyperparameters

However, the batch of images is first passed through the segmentation model, then the diseased portions of the batch are sent to the classification model. As before, the loss is computed, and parameter weights adjusted. Finally, a class-wise confusion matrix was computed.

Class	TP	FP	TN	FN	Precision	Recall
Bacterial Spot	451	258	4544	281	0.6361	0.6161
Early Blight	539	116	4775	104	0.8229	0.8383
Healthy	710	512	4221	91	0.5810	0.8864

Late Blight	594	150	4592	198	0.7984	0.75
Leaf Mold	591	134	4662	147	0.8152	0.8008
Septoria Leaf Spot	531	169	4619	215	0.7586	0.7118
Tomato Mosaic Virus	358	96	4854	226	0.7885	0.6130
Yellow Leaf Curl	299	26	5010	199	0.92	0.6004
MEAN					0.7651	0.7271

Additionally, the accuracy was determined to be 93.4%. The classification model's results depends on the results of the segmentation model. Thus, we believe improvements in the segmentation can greatly improve the classification.

Severity Calculation

The severity calculation uses pixel counting as mentioned in the design section. The number of diseased pixels is calculated by using numpy's count_nonzero function on the disease mask. Any non zero pixels are counted. The same approach is applied to the healthy mask. Finally, the severity is calculated using the formula.

$$Severity = \frac{disease_pixels}{disease_pixels + leaf_pixels}$$

Deployment

Finally, the models were deployed separately to the API since the basic Heroku dyno does not have sufficient RAM to load the models into memory. Since Azure offers free cloud functions for an unlimited time and does not have the same RAM limit, we chose to deploy our models there. This required using the Azure SDK and writing a script containing code for the inference pipeline.

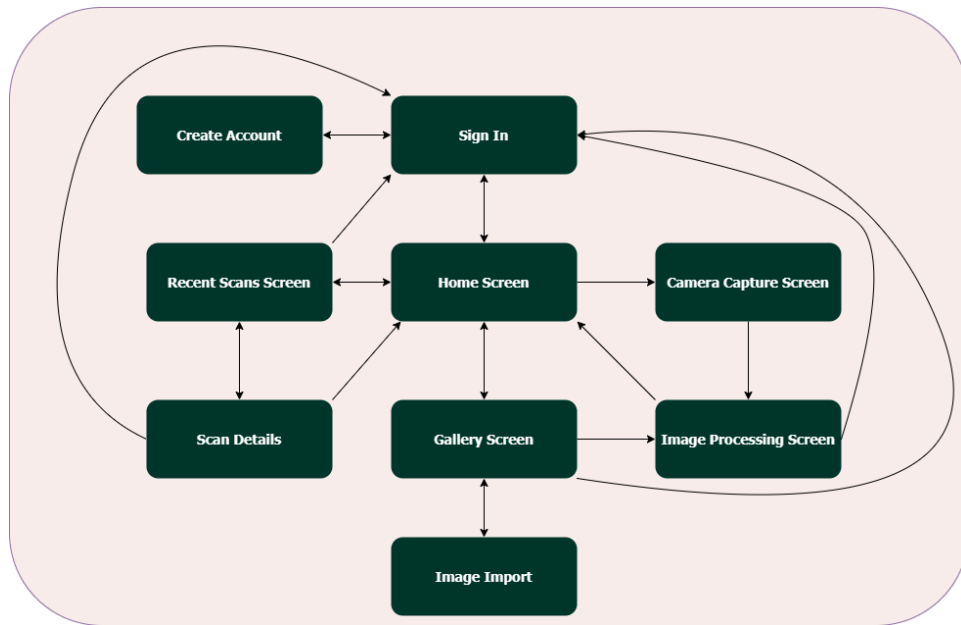
ANDROID APP

The application was developed in Android Studio using Java for implementing the Logic and xml for designing layout.

Key Libraries used:

- CameraX – CameraX is a Jetpack support library for Android that makes it easier to work with the camera hardware and capture photos or videos from within an Android app.
- OKHTTP - OkHttp is an open-source HTTP client library for Android and Java applications. It is designed to make sending and receiving HTTP requests as easy and efficient as possible.
- MediaStore - MediaStore is an Android framework that provides a centralized repository for media files such as images, videos, and audio. It allows apps to query, insert, update, and delete media files on the device, as well as listen for changes to media files in real-time. We used mediastore to save, retrieve and import on device images to and from the application.

Navigation Map:



Application Screens:

Create account and Sign in Screen

The image displays two side-by-side mobile application screens for an app named "Foliage Fixer". Both screens have a dark green background and a purple status bar at the top showing the time as 11:11 and 11:10 respectively, along with standard Android icons.

The left screen is titled "Create Account" and features three input fields: "Username" (with a person icon), "Password" (with a key icon and a toggle for visibility), and "Email" (with an envelope icon). A dark green "CREATE" button is positioned at the bottom center.

The right screen is titled "Sign In" and features two input fields: "Email" (with a person icon) and "Password" (with a key icon and a toggle for visibility). A dark green "LOGIN" button is positioned below the password field. At the bottom, there are two links: "Forgot Password?" on the left and "Create Account" on the right.

Home screen

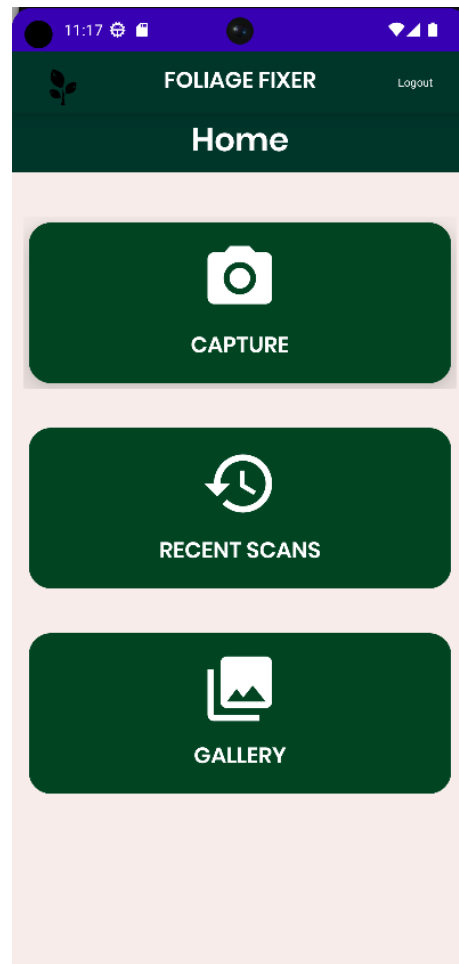
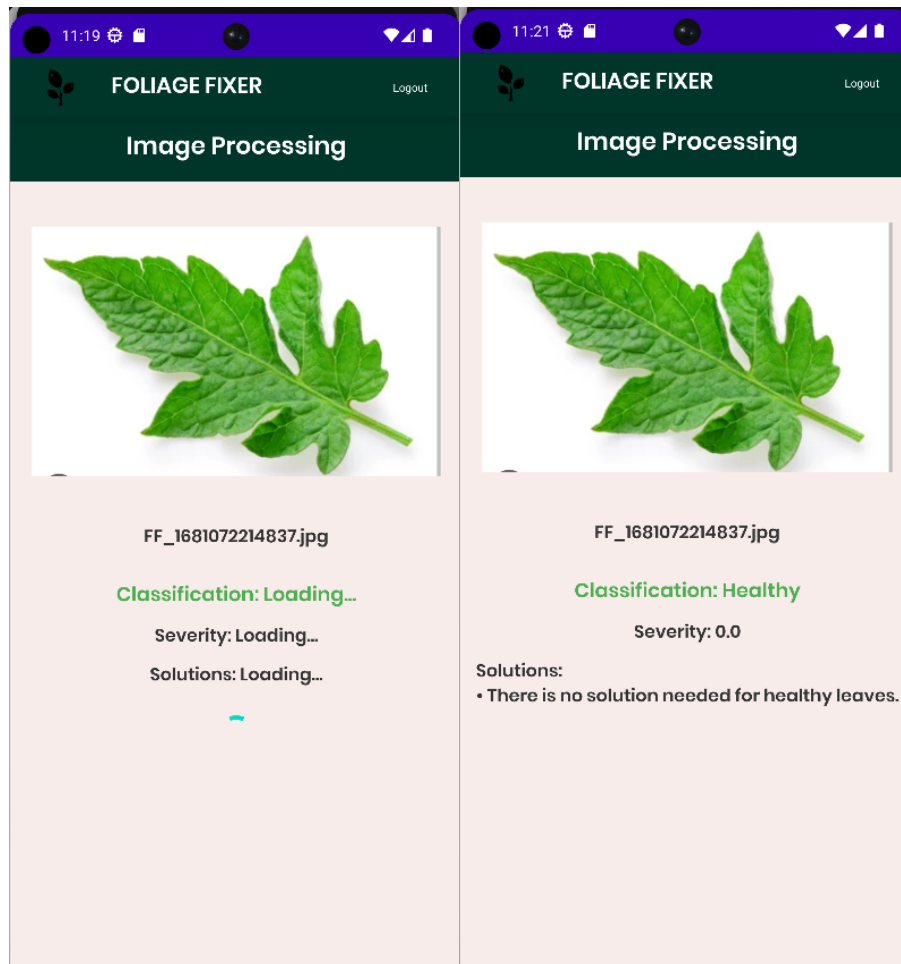


Image Processing




Recent Scans

11:36

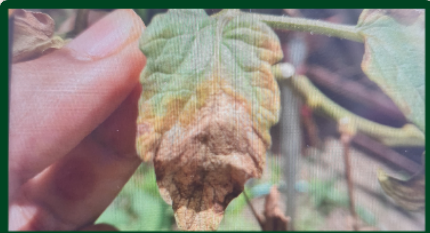
FOLIAGE FIXER

Logout


SCANS



Healthy
Severity: 0.10



Late Blight
Severity: 38.14




11:37

FOLIAGE FIXER

Logout

Scan Details

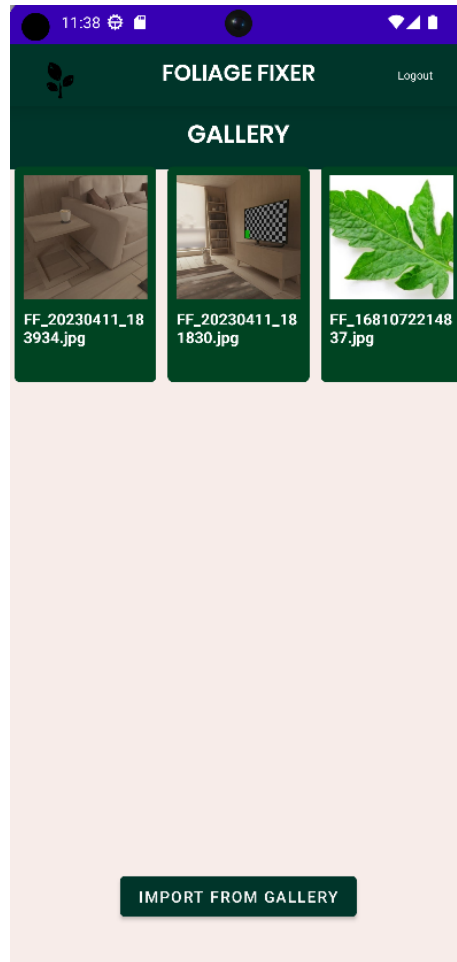


Late Blight
Severity: 38.14

Solutions:

- Apply a copper based fungicide (2 oz/ gallon of water) every 7 days or less, following heavy rain or when the amount of disease is increasing rapidly.

Gallery



TEST

Test Approach

For this project, our test approach consisted of unit testing, integration testing and acceptance and UI tests. The unit and integration testing was done in the flask API code using the python pytest library. The unit testing tested the individual classes and their methods. The integration testing tested the integrations between the API, the database and the azure function. Finally, the acceptance and UI testing was done on the Android app by following the test cases outlined in our test plan.

The application went through 3 major phases of acceptance tests.

Phase 1:

- Found compatibility issues with device storage usage with android 10 and newer
 - Changes: adopted use of scoped storage through Mediastore library as opposed to former implementation of storage read and writes.
- Compatibility issues with accessing the device's camera hardware on android 10 and newer.
 - Changes: switched to cameraX library rather than the now deprecated android camera library.
- Users indicated that there was too much steps to send image to process from importing an image into gallery
 - Changes: removed upload button from image processing screen so when an image is selected from the gallery, it would be instantly sent for processing and retrieve the results.

Phase 2:

- Users indicated that the gallery should have a delete function.
 - Changes: allow users to long press an image to delete it.
- Users indicated that they would like to be able select an image from the recent can list to view the image and the results in more detail.
 - Changes: added scan results screen

Phase 3:

- Users indicated that screen labeling was not consistent across different screens:
 - Changes: updated tool bar to remain consistent on all screens with the application title and logo. Added a secondary bar below the toolbar with screen titles.

Features Tested

The features tested include:

1. Classification
2. Segmentation
3. Image Capture
4. Loading from Gallery

TEST PLAN

Project Name: FoliageFixer

Test Cases

Test Case ID: Login_1

Test Priority (Low/Medium/High): Low

Module Name: Login

Test Title: Verify login with valid username and password

Description: Test the app's login page

Test Designed by: Ved Mahadeo

Test Designed date: April 1st 2023

Test Executed by: Ved Mahadeo

Test Execution date: April 10 2023

Pre-conditions: User has valid username and password

Dependencies:

Test No	Test Description	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes (optional)
TU01	User Login with valid Data	1. Navigate to login page	User= example@gmail.com	User should be able to login	User is navigated to dashboard with successful login	Pass	
		2. Provide valid username	Password: 1234				
		3. Provide valid password					
		4. Click on Login button					

Test No	Test Description	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes (optional)
TU02	User Login with invalid username	1. Navigate to login page	User example@gmiall.com	User should see error message for invalid user name.	As expected.	Pass	
		2. Provide invalid username	Password: 1234				
		3. Provide valid password					
		4. Click on Login button					

Project Name: FoliageFixer

Test Cases

Test Case ID: Signup_1

Test Priority (Low/Medium/High): Low

Module Name: Signup

Test Title: Verify signup

Description: Test the signup page

Test Designed by: Ved Mahadeo

Test Designed date: April 1st 2023

Test Executed by: Ved Mahadeo

Test Execution date: April 10 2023

Pre-conditions:

Dependencies:

Test No	Test Description	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes (optional)
TU01	User signup	Navigate to signup page	Email = example@gmail.com	User should be able to login	User is navigated to dashboard with successful login	Pass	
		Provide email, username, password	Password: 1234				
		Go to log in page.	Username = user				
		Enter created username and password.					

Project Name: FoliageFixer

Test Cases

Test Case ID: RecentScans_1

Test Priority (Low/Medium/High): Medium

Module Name: Recent scans page

Test Title: Verify recent scans load correctly

Description: Test the Google login page

Test Designed by: Ved Mahadeo

Test Designed date: April 1st 2023

Test Executed by: Ved Mahadeo

Test Execution date: April 10 2023

Pre-conditions: User example1@gmail.com should have more than 1 scans

Dependencies:

Test No	Test Description	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes (optional)
TU01	Get recent scans for new user with no scans	Sign up and log in	User= example1@gmail.com	Page is empty.	Page is empty.	Pass	
		Click recent scans.	Password = 1234				

Test No	Test Description	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes (optional)
TU02	Get recent scans for new user with 1 scan	Log in	User= example1@gmail.com	Page should have the recently captured image with the same	As expected.	Pass	
		Click "Image Capture"	Password = 1234				
		Capture image and wait for classification result.					

		Return to home, click recent scans.		classification and severity.			
--	--	-------------------------------------	--	------------------------------	--	--	--

Test No	Test Description	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes (optional)
TU02	Get recent scans for existing user with multiple scans	Log in	User= example@gmail.com	Page should have all the recently captured images for that user with the same classification and severity.	As expected.	Pass	
		Click “Image Capture”	Password = 1234				
		Capture image and wait for classification result.					
		Return to home, click recent scans.					

Test Case ID: classify_1

Test Priority (Low/Medium/High): High

Module Name: Classify

Test Title: Classification

Description: Test the app’s classification

Test Designed by: Ojore kanneh

Test Designed date: April 1st 2023

Test Executed by: Ojore kanneh

Test Execution date: April 10 2023

Pre-conditions: User has valid email and password

Dependencies:

Test No	Test Description	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes (optional)
TU0 1	Classify image from image capture	Navigate to login page	User= example@gmail.com	User should see new page with classification, severity and solution results	User navigated to new page with image of the leaf taken along with its classification, severity and solutions	Pass	
		Provide valid email and password	Password: 1234				
		Go to home page					
		Tap on capture					
Test No	Test Description	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes (optional)
TU0 2	Classify image from image gallery	Navigate to login page	User example@gmail.com	User should see new page with classification, severity and solution results	User navigated to new page with image of the leaf taken along with its classification, severity and solutions	Pass	
		Provide valid email and password	Password: 1234				
		Go to home page					
		Tap on gallery					
		Select image from gallery					

CONCLUSIONS, LESSONS LEARNED AND RECOMMENDATIONS

Since all planned deliverables, namely the android app, API and models were all completed we can say the project is fully complete. However, while the performance of the models is decent, it can be greatly improved.

Future Work

1. Improve classification and segmentation models – the segmentation model's performance is greatly impacted by the inadequate dataset size. Consequently, the classification model's performance is directly related to the segmentation performance as it functions as a preprocessing step. Therefore, by labelling a large amount of data both models' performance will improve.
2. Consider more diseases – The dataset used in this project omits 3 diseases from the original dataset. Additionally, tomato plants have many more diseases than just what is included in the dataset. Future versions of the app could include more diseases.
3. Consider more crops
4. Severity-sensitive disease management – The current system suggests the same solutions for a disease with 100% severity and a disease with 1% severity. Clearly, the management strategy will differ. Future work can include modifying the dataset of solutions to respond to sensitivity.
5. Better support for users with little to no wifi – It is very likely that users of the FoliageFixer will be using the app in the field, where wifi and data is very limited. While challenging, it is important to make accommodations for these users.

Lessons Learned

While the project was fully completed, our time management was not perfect. Most of the 12 weeks was spent learning about the CNN models and how to improve them. A smarter approach would've been to simultaneously finish core functionality of the app and API early on.

GIT REPO AND TRELLO

Trello

<https://trello.com/invite/b/18B9hT4L/ATTIdecb89ac391fcc8e9a8e208747a88a7d0D2326F3/foiage-fixer>

Model Training Repository

<https://github.com/Funky-UWI/FoliageFixerModel>

Deployed Model Cloud Function

<https://github.com/Funky-UWI/AzureFunctions>

Back End API

<https://github.com/Funky-UWI/FoliageFixerBackend>

Android App

<https://github.com/HariBarran/Funky-FoliageFixer-app>

ANDROID APK LINK

https://drive.google.com/file/d/1lnVvAtaPZNKE2XIUzQuvvO-t5PIOyqC6/view?usp=share_link

DEMO LINK

https://drive.google.com/file/d/112WPPLl4moz8fD-KW5sCe2KrtlqdsLPU/view?usp=share_link

REFERENCES

- Beron, Leidy Esperanza Pamplona, Andrés Felipe Calvo Salcedo, and Arley Bejarano Martínez. 2020. "Detection of Foliar Diseases Using Image Processing Techniques." *Revista Ceres* 67 (May): 100–110. <https://doi.org/10.1590/0034-737X202067020002>.
- Dawod, Rodica Gabriela, and Ciprian Dobre. 2022. "ResNet Interpretation Methods Applied to the Classification of Foliar Diseases in Sunflower." *Journal of Agriculture and Food Research* 9 (September): 100323. <https://doi.org/10.1016/j.jafr.2022.100323>.
- Gonçalves, Juliano P., Francisco A.C. Pinto, Daniel M. Queiroz, Flora M.M. Villar, Jayme G.A. Barbedo, and Emerson M. Del Ponte. 2021. "Deep Learning Architectures for Semantic Segmentation and Automatic Estimation of Severity of Foliar Symptoms Caused by Diseases or Pests." *Biosystems Engineering* 210 (October): 129–42. <https://doi.org/10.1016/j.biosystemseng.2021.08.011>.
- Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, et al. 2020. "Array Programming with NumPy." *Nature* 585 (7825): 357–62. <https://doi.org/10.1038/s41586-020-2649-2>.
- Hasty. 2023. "Hasty.Ai." <https://hasty.ai>.
- He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Deep Residual Learning for Image Recognition." arXiv. <http://arxiv.org/abs/1512.03385>.
- Lakubovskii, Pavel. 2019. "Segmentation Models Pytorch." https://github.com/qubvel/segmentation_models.pytorch.
- Noyan, Mehmet Alican. 2022. "Uncovering Bias in the PlantVillage Dataset." Medium. June 16, 2022. <https://towardsdatascience.com/uncovering-bias-in-the-plantvillage-dataset-7ac564334526>.
- Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." arXiv. <https://doi.org/10.48550/arXiv.1912.01703>.
- Ramzan, Farheen, Muhammad Usman Khan, Asim Rehmat, Sajid Iqbal, Tanzila Saba, Amjad Rehman, and Zahid Mehmood. 2019. "A Deep Learning Approach for Automated Diagnosis and Multi-Class Classification of Alzheimer's Disease Stages Using Resting-State FMRI and Residual Neural Networks." *Journal of Medical Systems* 44 (December). <https://doi.org/10.1007/s10916-019-1475-2>.
- "Tomato Disease Multiple Sources | Kaggle." n.d. Accessed April 15, 2023. <https://www.kaggle.com/datasets/cookiefinder/tomato-disease-multiple-sources>.
- TorchVision maintainers and contributors. 2016. "TorchVision: PyTorch's Computer Vision Library." <https://github.com/pytorch/vision>.
- Wang, Zifu, and Matthew B. Blaschko. 2023. "Jaccard Metric Losses: Optimizing the Jaccard Index with Soft Labels." arXiv. <http://arxiv.org/abs/2302.05666>.
- Zhou, Zongwei, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. 2018. "UNet++: A Nested U-Net Architecture for Medical Image Segmentation." ArXiv.Org. July 18, 2018. <https://arxiv.org/abs/1807.10165v1>.