# CHAPTER 3: METHODOLOGY

## 3.1.1 Introduction

This chapter explains in detail the stages involved in implementing the proposed disease detection system for crop leaves.

## 3.1.2 Implementation Overview

The program will be used to monitor tomato plants on a farm, identifying infections and diseases by scanning an image, analyzing it, and then evaluating whether or not the plant is afflicted and how severe the condition is.

The implementation of the system involves the following stages:

* Data Collection
* Data Pre-processing
* Building the CNN Model
* Backend Fast API TensorFlow
* Website (React JS)
* Mobile App (React JS)

## 3.2.0 Data Collection

The dataset used for training and testing the model is a tomato leaf dataset from Kaggle.com. The dataset contains tomato leaves with early blight and late blight as well as fresh leaves [34].

## 3.3.0 Data Pre-Processing

### 3.3.1 Setting Constraints

Text

Description automatically generated

Figure 1

We set constants such as epochs, image sizes, RGB channels, batch size as well as the directory containing the dataset to be used.

### 3.3.2 Initializing & Partitioning the Dataset

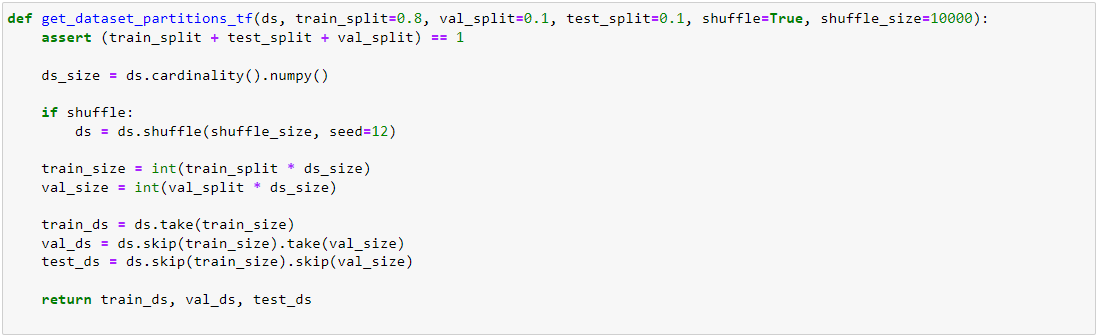


Figure 2

The dataset is split into 3 subsets, which are; training, validation and test. The training dataset will be used during training, the validation dataset to be tested against while training and also the test to be tested against after model training. We create a TensorFlow Dataset Object and directly read it from the directory using `image\_dataset\_from\_directory` and then split it using the function we created above.

Word

Description automatically generated with low confidence

Figure 3

We then use the above function to display the class names or into which the images are categorized.

## 3.3.0 Building The CNN Model

### 3.3.1 Resizing and Normalisation Layer

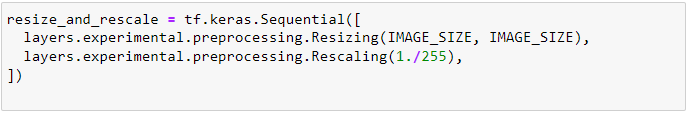


Figure 4

Before feeding images into the network, we must resize the images to our preferred size. Moreover, we should normalize the image pixel value (keeping them in range 0and 1 by dividing by 256) to optimize model performance. This is done during training as well as inference. Therefore, it can be added as a layer in our Sequential Model.

### 3.3.2 Data Augmentation

Graphical user interface, text

Description automatically generated

Figure 5

Data Augmentation is needed when we have less data, this boosts the accuracy of our model by augmenting the data. After data augmentation we also check the expected dimension order for the channel as shown in figure 6 below.

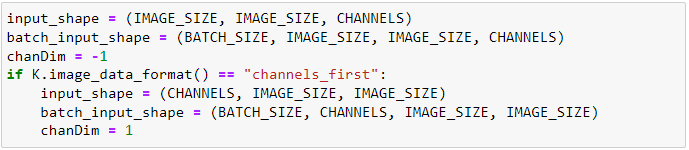


Figure 6

### 3.3.3 Model Architecture

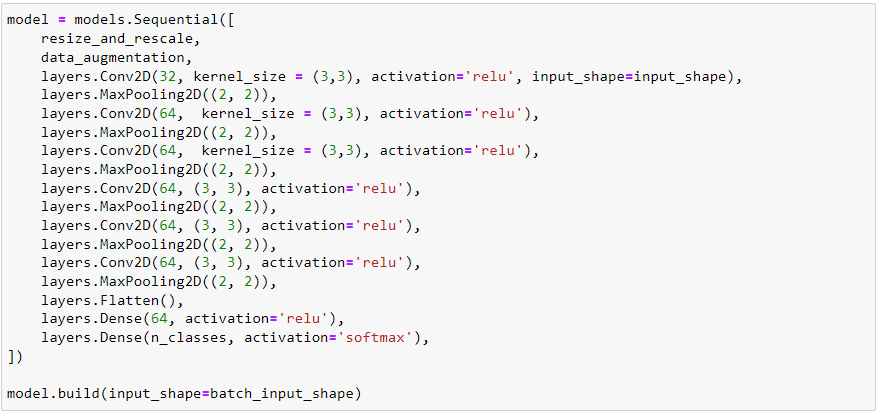


Figure 7

We use a CNN coupled with a SoftMax activation in the output layer. We also add the initial layers for resizing, normalization and Data Augmentation.

### 3.3.4 Compiling the Model

Graphical user interface, text

Description automatically generated

Figure 8

We use the *Adam* Optimizer*, Sparse Categorical Cross entropy* for losses and *accuracy* as the metric.

### 3.3.5 Training the Model

Graphical user interface

Description automatically generated with low confidence

Figure 9

The model is then trained on the training dataset and the set parameters such as epochs and defined batch size are used in doing so.

### 3.3.6 Testing the Model

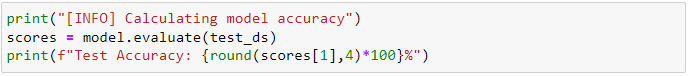


Figure 10

After training we test the model on the predefined test dataset and thus determining the accuracy of the model.

### 3.3.7 Inference Function

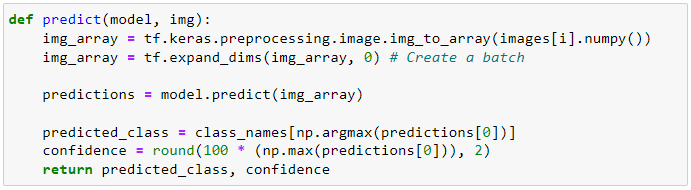


Figure 11

The inference function will be called when an image of a tomato plant leaf is uploaded for scanning purposes as shown above in figure 11.

### 3.3.8 Displaying Inference Data

A picture containing chart

Description automatically generated

Figure 12

The data inferred is plotted, indicating the actual uploaded picture, then the predicted disease and also the prediction confidence.

## 3.4.0 Backend Fast API with TensorFlow

A fast API server model was implemented such that both the website and mobile application can access and use the model for inference and predicting or detecting infections on the tomato plant leaves.

### 3.4.1 Required Libraries for Fast-API

Text

Description automatically generated

Figure 13

Figure 13 shows the required libraries to run the Fast-API server, which include:

* TensorFlow==2.5.0
* Fast API
* Uvicorn
* Python
* Pillow
* Matplotlib
* Numpy

### 3.4.2 Server URL’s and Endpoint

Graphical user interface, text, application

Description automatically generated with medium confidence

Figure 14

Figure 14 shows the server URL’s as well as defining the endpoint for the API.

### 3.4.3 Image Upload



Figure 15

As shown in figure 15, this section of the API is responsible for the file upload feature, which will be used in uploading an image for the model to determine if the leaf shown in the image is infected. This section also returns the predicted disease classification (such as Early-blight, Late-blight or Healthy leaf) and also the confidence of the predication in JSON format, which will then be transferred to our user interface for the website and mobile application.

## 3.5.0 Website (React JS)

Diagram

Description automatically generated

Figure 16

Figure 16 shows the interface for the web platform that one can use to check whether a leaf is diseased or not. The image is uploaded by simply dragging an image of a tomato plant leaf and the leaf will be processed as shown below in figure 17.

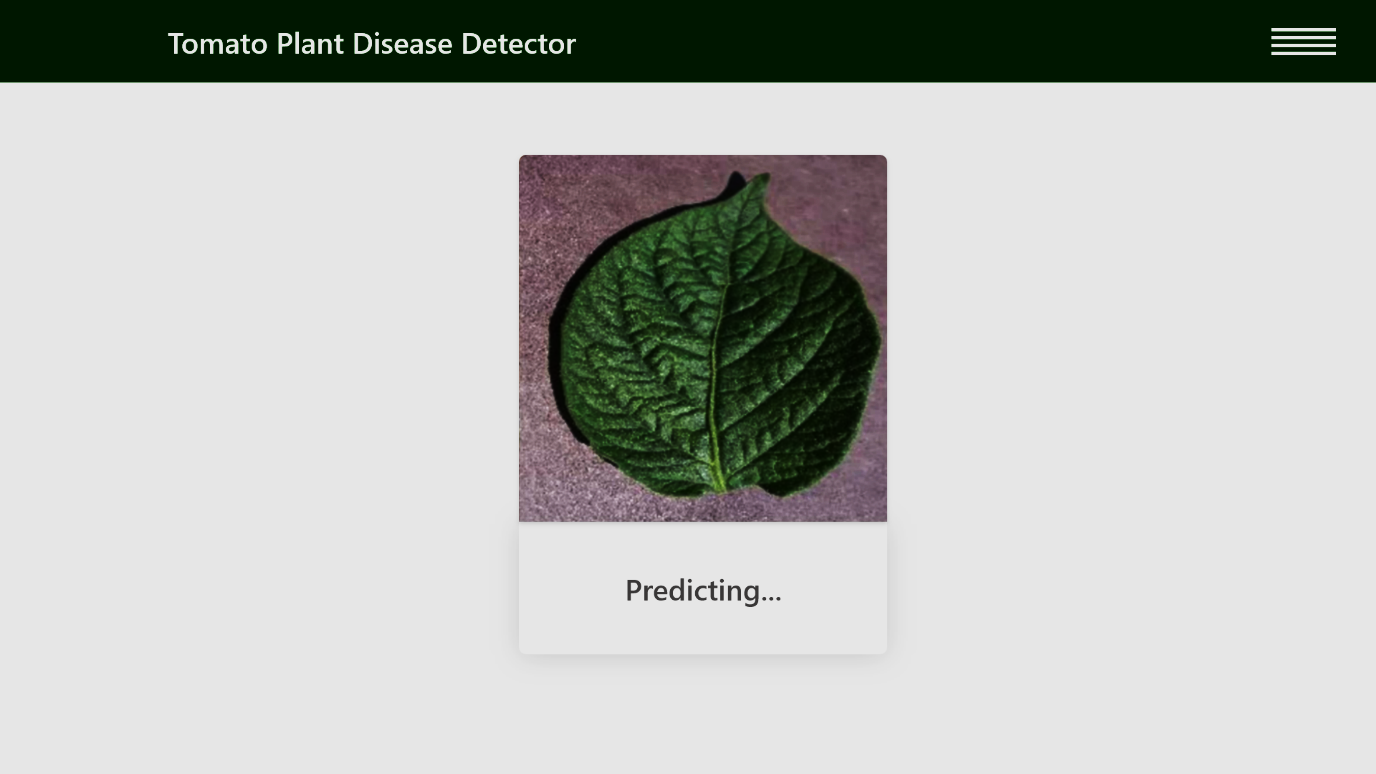


Figure 17

Figure 17 shows the predicting state illustrated by the above screenshot on the website.

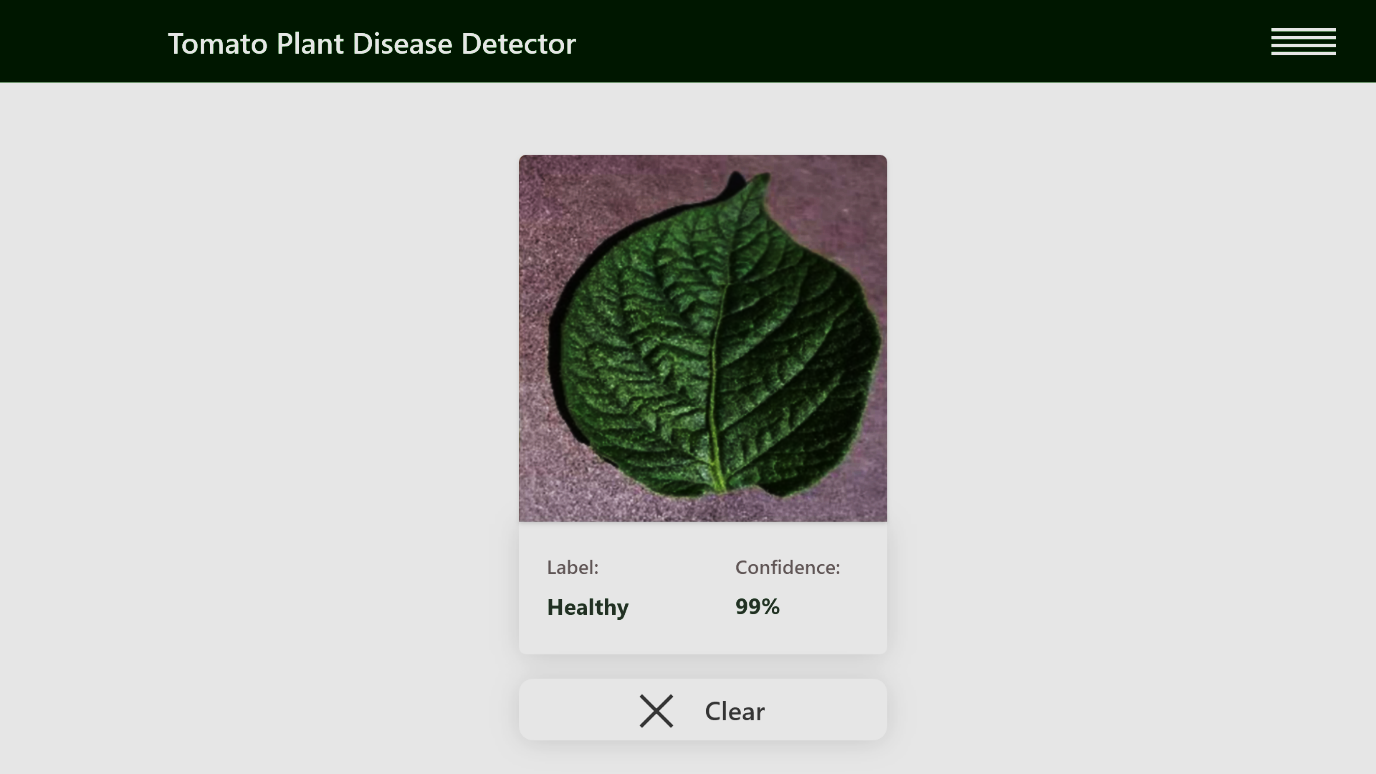


Figure 18

On figure 18, the leaf has been scanned and analysed. The label section shows the classification of the leaf uploaded by the user (Healthy, Early Bight or Late Blight). The confidence is also displayed, the confidence in this case is the accuracy of the model on this particular leaf test. A Clear button is also available to refresh the site and take us back to the home page.

### 3.6.0 Mobile Application (React JS)

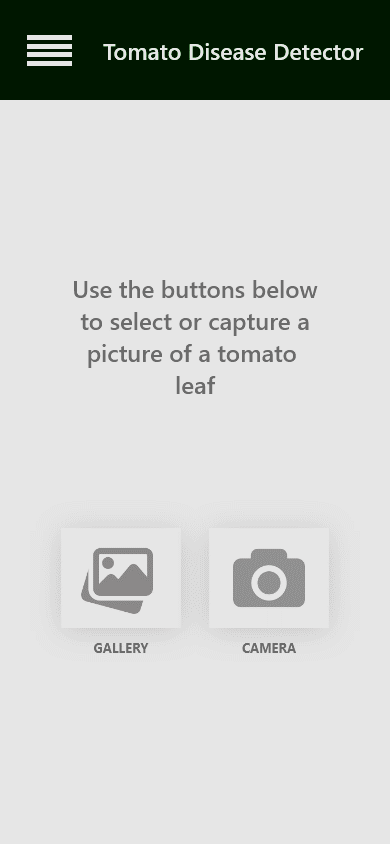
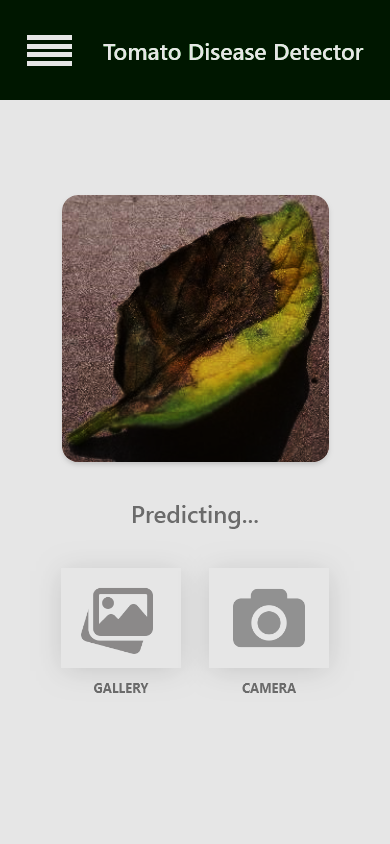
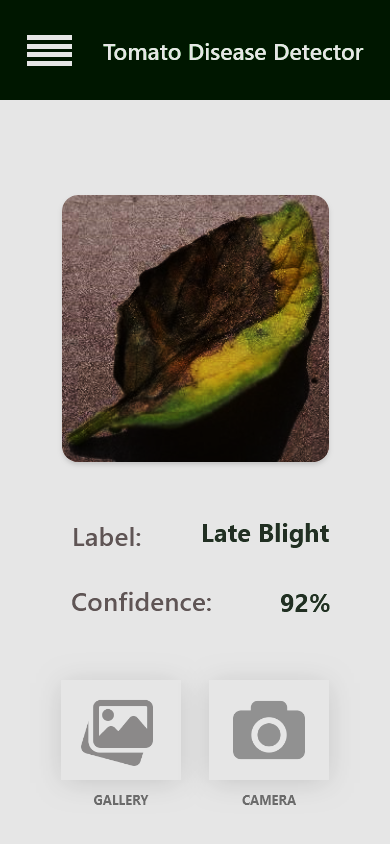
  

Figure 19

Figure 19 shows screenshots of the 3 phases involved when using the mobile application. The user chooses whether to select an existing image of a tomato leaf, or alternatively take a photo from the mobile device’s camera. The photo is then transferred via HTTP to our Fast-API server to enable the model to access the image. After inference by the model, the responses are returned to the user interface and the user can see the Label or Classification as well as the accuracy of the prediction.

### 3.7.0 Overall System Architecture

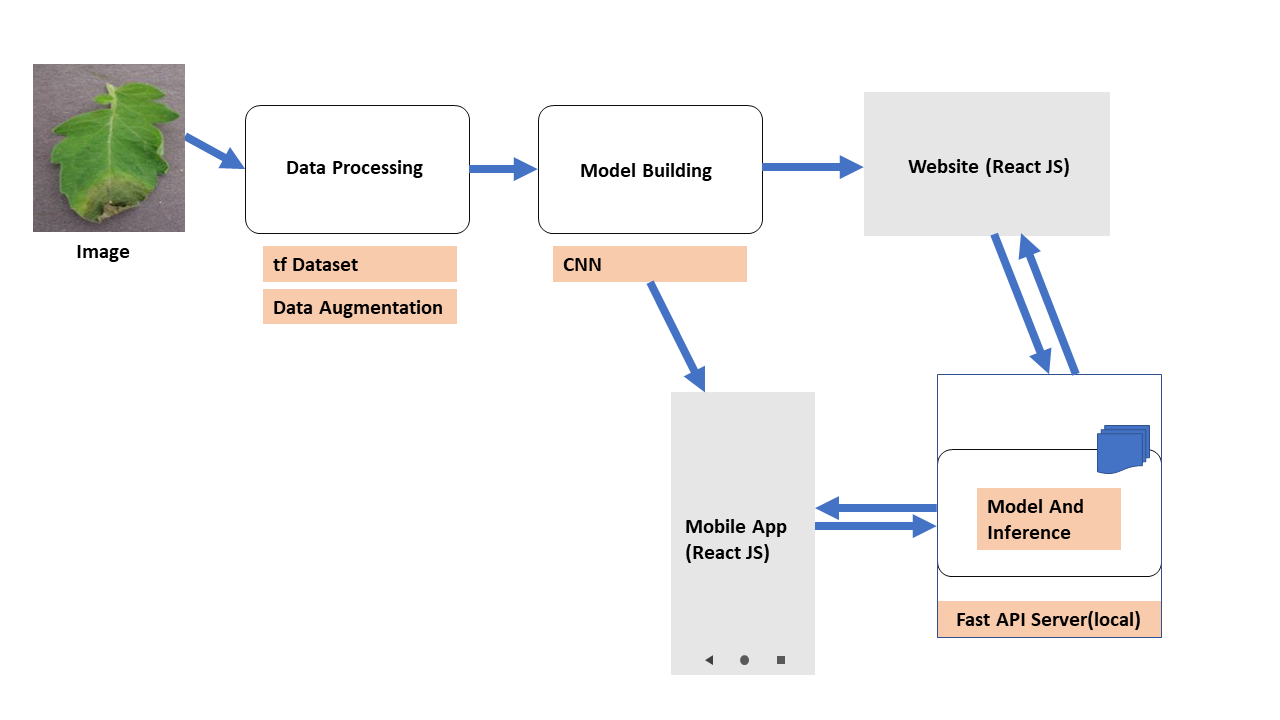


Figure 20

The overall system architecture for both the mobile application and the website are identical in that the inference model is being accessed in the same way and typically involves identical processes except that one is a mobile interface whilst the other is a web interface. The overall process involves having a dataset to begin with. We then split this dataset into training and testing. Afterwards the data is cleaned and augmented and thereafter we build and train our model with the training data. The website and mobile applications will be used by users to upload their images and then inference is done on the images and the predicted disease and confidence (accuracy) is returned and displayed.

### 3.8.0 Summary

The chapter showcased how the plant disease detection was implemented as well as the different components and their roles in the system. The next chapter will review the results of the system and models used for this system.