

Ikobi Lynch

Ayodeji Adedipe

Giovanne Pinto

Rajahni Cunningham

Contents

[Introduction 5](#_Toc135343041)

[Problem Statement 5](#_Toc135343042)

[Requirements 5](#_Toc135343043)

[Functional Requirements 5](#_Toc135343044)

[Non-Functional Requirements 5](#_Toc135343045)

[This Project 6](#_Toc135343046)

[What we have accomplished 6](#_Toc135343047)

[Background 7](#_Toc135343048)

[Technologies used 7](#_Toc135343049)

[Similar technologies 7](#_Toc135343050)

[Method 8](#_Toc135343051)

[Architectural Design 8](#_Toc135343052)

[Use Case Diagram 8](#_Toc135343053)

[Use Case: Upload Plant Image 9](#_Toc135343054)

[Use Case: Upload Plant Image 9](#_Toc135343055)

[Architectural Overview 11](#_Toc135343056)

[View 11](#_Toc135343057)

[Controller: 13](#_Toc135343058)

[Model 16](#_Toc135343059)

[Understanding the requirements 17](#_Toc135343060)

[Data Understanding 17](#_Toc135343061)

[Data Preparation 17](#_Toc135343062)

[Modelling 18](#_Toc135343063)

[Evaluation 26](#_Toc135343064)

[Deployment 28](#_Toc135343065)

[TensorFlow Category Set: 28](#_Toc135343066)

[Scalability 28](#_Toc135343067)

[Results 29](#_Toc135343068)

[Project Success 29](#_Toc135343069)

[Conclusion 32](#_Toc135343070)

[Project Assessment 32](#_Toc135343071)

[Alternative Approach 32](#_Toc135343072)

[Lessons 33](#_Toc135343073)

# Introduction

## Problem Statement

Plants play a fundamental role in supporting life on Earth by providing essential resources such as food, oxygen, and medicinal benefits. Due to the sheer vastness of our planet, there exists a wide variety of plants and plant species, each possessing distinct characteristics that are unique to their species. Accurately identifying these plant species is crucial for various fields, including agriculture, ecology, and medicine.

Traditionally, the identification of plant species has relied upon physical characteristics such as leaf shape, leaf size, and flower colour. The methods used to identify plants manually can be time-consuming, require specialized knowledge, and are prone to errors. The advances in computer vision and machine learning, have enabled the development of automated image identification systems. These systems have been beneficial to plant identification, making the process faster, more accurate, and accessible to wider range of people.

Requirements

Functional Requirements

Based on the problem to be solved, the requirements of the system are:

1. The system shall accept a plant image from users.
2. The system shall perform image processing in which an uploaded image is pre-processed to standardize its quality and size.
3. The system shall analyze the preprocessed image using the deep learning model.
4. The system shall classify the image.
5. The system shall predict the most likely species of the uploaded image.
6. The system shall give feedback on the name of the plant.

Non-Functional Requirements

* The system shall have a high accuracy rate and therefore a low error rate.
* The system shall respond promptly to interactions; there must be minimal lag between a command being made and followed
* The system shall generate a prediction of the plant within 10 seconds of the **Generate Prediction** button being clicked.
* The system shall be intuitive, and easy to use, with a bias to words instead of icons.
* The system shall be able to handle an increase in the dataset over time without loss in performance or accuracy.

This Project

This capstone project aims to contribute to the ongoing efforts to develop a plant identification system that can accurately recognize plant species from digital images of the plant's leaves. The system could have many real-world applications in various areas such as horticulture where it can be used to aid in determining a plant’s health based on its leaf’s appearance; in plant research where being able to identify a plant quickly is crucial; in identifying plants and their medicinal benefits, and even non-specialist applications, such as a layman being curious about a plant that they have seen.

## What we have accomplished

The proposed system employs a convolutional neural network (CNN), which is a type of deep learning algorithm that can be used for image recognition tasks. The CNN model is trained to extract the unique features of a leaf from images in a dataset. The model can analyse a leaf’s properties, distinguish the patterns in the leaf such as veins, apexes, and midribs, and make computations based on these characteristics of the leaf. As it is presented with more images from the training dataset, it can “learn” more about each plant and determine its specie. After these computations are made, the details are passed to a classification algorithm to identify the plant’s specie.

To make a reference to the requirements listed, the current system to date:

* allows an image to be uploaded by a user,
* allows an image to be processed by the CNN,
* returns a label detailing the name of the plant species in the photo as well as the condition of the plant based on the leaves,
* is trainable should more images be added to the dataset,
* achieves an above 95% accuracy in predicting the correct plant given based on those leaves used in the test dataset given.

# Background

## Technologies used

In developing our system, we employed the use of pre-existing tools and technologies. Some notable tools we used are Python, Streamlit, TensorFlow and Adobe Illustrator.

Python was used as the base language to implement the model. To create the functions and methods necessary to implement and train a convolutional neural network, libraries from TensorFlow were used that were designed for this purpose. Within these libraries are modules used to specify the type of neural network, the functions of the layers within the network, the methods by which the images data should be handled by the neural network, and the optimisation methods that should be used during the training process.

JavaScript was the chosen language used in creating the accompanying marketing website for the system. It was selected due to its versatility in creating website functionality and was proven capable of handling what was needed for this website. JavaScript also integrates well with both HTML and CSS.

HTML was used to create the different elements needed for the website’s layout and was chosen because of its extremely high compatibility with most popular browsers such as Google Chrome, Mozilla Firefox, Microsoft Edge, and Apple Safari.

CSS and Bootstrap were used together to give the website an attractive and easy-to-use layout which would enhance a user’s experience while using the website.

Adobe Illustrator was used to create a vector image logo to accompany the system.

Streamlit.io was used to port the system to a web-interface so that a user may interact with the system. It is through this system that a user can upload an image, and subsequently have the name of the plant returned.

Similar technologies

Similar pre-existing technologies exist such as those found on [Pl@ntNet identify (plantnet.org)](https://identify.plantnet.org/). This system performs in a very similar way, however a benefit that the “Herbal AI” system has is the ability to further train the system to specialise in identifying plants from a certain region, country or area which enables it to find identify plants that are indigenous to that area. For example, it will be possible to train the AI on plants that are indigenous or local to Jamaica. The system also possesses the ability to determine leaves with certain diseased spots which makes it beneficial to personnel within a field that require a certain knowledge.

# Method

### Architectural Design

This section will showcase and describe the main structural components in the system, in other words, the overall design of the system as it relates to constructing the main diagrams for the system. The relationship between these diagrams/designs illustrates the core of the Herbal AI system to be constructed. Though the aim was not to build the system within a specific architectural design realm, many elements from the Model-View-Controller (MVC) design pattern were implemented. This pattern distinguishes components or modules based on the functional roles of each part. The **Model** holds stored data for the system to use. It acts as a database of sorts. The Controller is the section that intervenes between the user and the system and performs majority of the logical functions of the system. The View is the component that provides the user interface.

### Use Case Diagram

The use case diagram for this system is rather simple as the two main functions of the system are to import an image of a plant, and to return the name of the plant to the user.

A screenshot of a computer screen

Description automatically generated with medium confidence

Figure 1: use case diagram

### Use Case: Upload Plant Image

The system allows a user to upload an image of the leaf of the plant they wish to know the name of. The image is loaded in the system. The system does some pre-processing and resizing to the image and shows a preview of the image to the user. The options to generate the prediction is then given to the user. The user then clicks “Generate prediction” which begins the image processing. After the image has been processed the system returns a label stating the name of the predicted plant.

Primary Actors:

* The user: This is an individual or organisation who wishes to identify a plant species. The user.
* The image upload system: The system imports the image and processes the data.The trained, automated system that will be used to identify the plant using AI classification.

Preconditions:

* The user should have access to the system through the web application.
* The user should possess a digital image of the plant’s leaf. The image should be clear and contain a whole leaf.

Basic flow of events:

1. The user opens the plant identification system via the web application.
2. The user selects the “Browse” option or drag and drop an image of the leaf, to identify a plant species from the available menu.
3. The system resizes, and does some conversions to make the image suitable for the model.
4. The system shows a preview of the image on the user interface.

### Use Case: Upload Plant Image

The user then clicks “Generate prediction” which begins the image processing. After the image has been processed the system returns a label stating the name of the predicted plant.

Primary Actors:

* The user: This is an individual or organisation who wishes to identify a plant species. The user will be the one receiving the plant name information from the interface.
* The Model: The system imports the image and processes the data. The trained, automated system that will be used to identify the plant using AI classification.

Basic flow of events:

1. The user selects “Generate prediction”.
2. The system receives the image and initiates the analysis process.
3. The system employs the convolutional neural network (CNN) to extract unique features from the image.
4. The extracted features are then processed through a classification algorithm to determine the plant species.
5. The system presents the results to the user, displaying the identified plant species.

*To gain further understanding of the processes that occur in the background, an overview of how the system was implemented will be explained.*

Architectural Overview

The plant identification system borrows elements from a MVC design pattern type Architecture. The parts are separated into three categories: View, Controller, and Model. Initially the design was not thought to be constrained to this type of design, however after assessing the parts of the system as well as the way in which they were integrated, it became apparent that it was like the MVC pattern. This is seen where we have an application serving as the View component which is a user interface to interact with the system. The Controller module of the system include the components that import an image into the system and does pre-processing on the image to convert and resize it so that it is suitable for the CNN to use. The final module, the Model, contains data such as the trained CNN model that the system uses to make its predictions, and the classification algorithm that produces the output to be sent to the user interface.

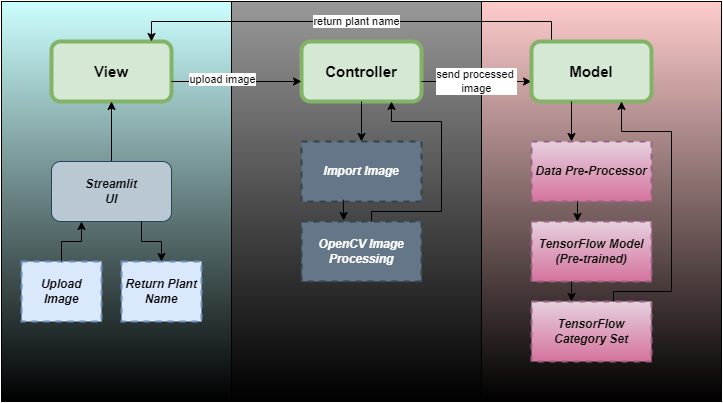


Figure 2: high-level layered architecture

View

The View component was built using the Streamlit library. This was chosen as it is a lightweight application which makes it ideal to run across multiple device types. The library also has modules to make transitioning from a web-application to a mobile application easier.

* **Streamlit User Interface (Streamlit UI):** This component uses the Streamlit Python library to create a user interface application for the system. The application allows a user to upload an image of type “jpg”. The user can do this by dragging and dropping an image to the page, or by clicking the “Browse files” button and selecting the required image from their local device. After uploading an image, the user gets to see a preview before moving on to the next step. The user interface then displays a button, “Generate Prediction” for the user to click. Once this is clicked, the system uses the saved model in conjunction with the CNN, and a dictionary of classifications to make a prediction about the type of leaf shown in the image. Finally, after processing, the plant’s name is returned to the user through the user interface.

Controller:

This section can be broken down into two separate parts that are responsible for pre-processing the image to be sent to the **Model** component. These are:

* **Import Image:** This is responsible for collecting the image data that is to be passed to the model. This subcomponent reads the uploaded files and converts it to a bytearray format. The data is then converted to a NumPy array which specifies that each element in the array should be an 8-bit unsigned integer type. This step is necessary to prepare the file for further processing and to increase the efficiency of data processing because:
  + NumPy is a powerful numerical computing library in Python that provides efficient array operations and mathematical functions. By converting the image to the NumPy format, it becomes compatible with NumPy's array-based operations, allowing for efficient numerical computations on the image data. This is particularly useful for tasks such as resizing, normalization, data augmentation, and other pre-processing steps.
  + Integration with machine learning frameworks: Many machine-learning frameworks, including TensorFlow, use NumPy arrays as the standard input format for training and inference. By converting the image to the NumPy format, it becomes easier to integrate with these frameworks and feed the image as input to machine learning models. Additionally, the NumPy format enables the utilization of various data augmentation techniques.
  + Interoperability with other data processing libraries: NumPy arrays can seamlessly interface with other data processing libraries and tools in the Python ecosystem. For example, the converted NumPy image can be easily integrated with libraries like **matplotlib** for further image analysis, manipulation, visualization, and processing. This interoperability allows for a wider range of image processing and analysis techniques to be applied to the image data.
  + Compact representation and memory efficiency: NumPy arrays provide a compact and efficient representation of multi-dimensional data. By converting the image to the NumPy format, it is stored as a multi-dimensional array, which optimizes memory usage and allows for efficient storage and manipulation of image data. This is particularly beneficial when working with large datasets or when memory resources are limited as is the case seen in developing this system.
* **OpenCV Image Processing:** This component receives the Numpy array data taken from the **Import Image** sub-component. Next, the system decodes the 8-bit bytearray representation of the image and converts it to an Open-Source Computer Vision Library (OpenCV) image format. The benefits of converting the image to this format lies in its operations and functions that make using this system more efficient. These benefits in detail are:
  + Compatibility with OpenCV: OpenCV is a widely used open-source computer vision and image processing library. By converting the image to the OpenCV format, it becomes compatible with the functions and operations provided by OpenCV. This allows for further image processing, analysis, and manipulation using OpenCV's extensive set of tools and algorithms.
  + Efficient image manipulation: OpenCV provides efficient and optimized functions for image manipulation tasks such as resizing, cropping, colour space conversion, and filtering. By converting the image to the OpenCV format, it becomes easier to leverage these functions and perform various image processing operations efficiently.
  + Consistent colour representation: The image is converted from its original format to the OpenCV format, which typically represents images in the Blue-Green-Red (BGR) colour order. This ensures consistency in the colour representation, as different image formats can have varying colour channel orders. By converting to the BGR colour order, subsequent image processing operations and display functions can work reliably with the image's colour channels.
  + Streamlit integration: In the given code, the image in the OpenCV format is displayed in the Streamlit user interface using the Streamlit.image() function. Streamlit provides native support for displaying OpenCV images, making it convenient to visualize and interact with the processed images within the Streamlit application.

After the image has been converted to the OpenCV format, the system displays a preview of the image. Before this image is displayed on the user interface, the image is converted to a format more suitable for viewing. This is accomplished by converting the colour space of the image from the BGR format that it is in, to a Red-Green-Blue (RGB) format. This conversion serves the purpose of:

* Visualization: RGB is the colour space that most displays and screens use to represent images. Converting the image to RGB ensures that it is displayed accurately and as intended when visualizing the image in applications, user interfaces, or other outputs. It allows for proper visualization and interpretation of the image by humans.
* Perception of colours: The order of colour channels in the BGR colour space is different from the RGB colour space. Converting to RGB aligns the order of colour channels with the human perception of colours. It allows for intuitive understanding and interpretation of the image's colour information by humans and ensures that the colours are represented correctly when viewed by humans.

The next function within this subcomponent resizes the image to specific dimensions of 224\*224 pixels. This step is necessary to ensure that the input image matches the required size expected by the CNN model. The specific purposes of this resizing are as follows:

* Standardisation: Resizing the image to a specific size ensures that all input images have the same dimensions. Standardising the image size is important for maintaining consistency in the input data across different images. This is particularly crucial for CNN models that expect fixed-size inputs, as it enables them to process the images uniformly.
* CNN compatibility: Machine learning models, including convolutional neural networks (CNNs), often have specific input size requirements. Resizing the image to the required dimensions ensures that it can be fed into the model correctly. In the given code, the image is resized to (224 \* 224), which is a common input size for many pre-trained CNN models.
* Performance optimization: Resizing the image can help optimize the computational performance of the model. Larger images require more computational resources and processing time. By resizing the image to a smaller size while maintaining the essential information, the computational burden on the model is reduced, resulting in faster inference and improved overall system performance.
* Conserving memory: Resizing the image can also be beneficial for memory conservation, especially when dealing with large datasets or limited memory resources. Smaller-sized images occupy less memory, allowing for efficient storage and processing of image data.

These functions are responsible for pre-processing the image and preparing it for further processing, as well as displaying the uploaded image to the user in the correct format. After this pre-processing occurs, and the image preview has been displayed, the next step for this component is to pass the resized and pre-processed image data to the pre-trained model to begin the classification process.

Model

The **Model** component of the system contains majority of the system’s logic and the functions necessary to make a prediction. This part of the system takes data from the **Controller** and tests the image against the pre-trained model. It is responsible for analysing the pixel data within the image, making comparisons based on the known classified images in the dataset contained in the model, and giving predictions for each other possible classifications that are possible. It then returns an array of all the probabilities for each plant. The system makes the final prediction by choosing the array element with the highest probability and then referencing the element’s index with the plant classes present in a predefined dictionary. For example, if the plant with the highest probability is at index position two (2) of the array, the system checks what dictionary key two (2) has as its value and returns that to the user as the label for the image. The breakdown of the individual processes is as follows:

* Data Pre-processor: This part of the system uses the “tensorflow.keras.applications.mobilenet\_v2” module to run further pre-processing on the image data. It is implemented as the first layer of the CNN and it was a design decision made to increase the accuracy and efficiency of the model and therefore the accuracy of the predictions made. The module does it’s pre-processing in steps. These are:
  + Channel-wise mean subtraction: The function subtracts the mean RGB pixel values from the input image. The purpose of this operation is to centre the pixel values around zero and remove any bias that might be present in the dataset. This involves:

1. Calculating the mean value for each colour channel across the entire dataset. This involves computing the average value of all pixel intensities within each channel.
2. Subtracting the computed mean value of each channel from the corresponding pixel values in the image. This operation is performed separately for each colour channel.

This helps in reducing any systematic bias present in the dataset and improves the performance and convergence of the CNN model during training and inference.

* + Normalization: The function normalizes the pixel values by dividing them by 255. This step scales the pixel values to a range between -1 and 1, making them suitable for processing by the MobileNetV2 model. The pixel values being normalised ensures that each colour channel has similar scale and importance. In other words, it prevents one channel from dominating over the others.
* TensorFlow Model (Pre-trained):

This subcomponent contains the backbone of the entire system. The model trained to identify plants is saved here. To train and develop this model, many different tasks had to be executed and in a stepwise manner. These steps are adopted from the CrispDm methodology:

Understanding the requirements

To begin the data mining process, the first step was mainly about determining and understanding the requirements needed for the project. In doing so we had to assess the purpose of this system to an end-user. It was understood that we needed a lightweight, easy to use, and accurate system that would allow a user to upload an image of a plant’s leaf and have the name of the plant returned to them. After this was comprehended, the next goal was to determine where to find the data needed to train the model. At this phase of the development, the main task was locating a large dataset containing plant images and their accompanying labels. This proved rather difficult, as there are not many plant datasets with a large variety of plants publicly available; however, the datasets found contained images of leaves with different types of diseases. Though not the ideal dataset, it was usable to the proposed model. This is because the model needs to learn to identify subtle differences in a leaf’s pattern, and these images contained enough variations for the model to be trained without overfitting.

The next task was to determine the correct tools to use to build and train the model. Through research, it was decided that a good solution was to employ the use of TensorFlow libraries.

Data Understanding

This and the next phase of the development proved to be two of the more time-consuming parts due to the data that was available. The team selected the “plant\_village” dataset and the plant\_leaves dataset that was available through TensorFlow. The plant\_village dataset contains 54,303 files in total. The plant\_leaves dataset contains 4502 files. The files from both dataset’s images are stored in JPEG format. The categories were determined by the name of the folder that the image is in, within the respective datasets. These categories served as the labels for the images.

Data Preparation

To reduce the chance of overfitting, and to give the model more variety to train from, both datasets were combined to create a new dataset. The new dataset is called “Plant\_dataset”. This dataset contains a total of twenty-five (25) categories. The categories were created in a similar format to those in the original datasets, in that, the label for the images were determined based on the name of the folder that contained the image. Through this, a new and unique dataset was created to handle the training and testing processes. These categories are:

|  |  |
| --- | --- |
| Cherry (Powdery Mildew) | Squash (Powdery Mildew) |
| Corn (Common Rust) | Strawberry (Leaf Scorch) |
| Corn (Healthy) | Tomato (Bacterial Spot) |
| Grape (Black Rot) | Tomato (Early Blight) |
| Grape (Esca aka Black Measles) | Apple (Cedar Apple Rust) |
| Grape (Isariopsis Leaf Spot aka Leaf Blight) | Apple (Black Rot) |
| Orange (Huanglongbing aka Citrus Greening) | Apple (Healthy) |
| Peach (Bacterial Spot) | Apple (Scab) |
| Pepper Bell (Bacterial Spot) | Blueberry (Healthy) |
| Pepper Bell (Healthy) | Pomegranate (Healthy) |
| Potato (Early Blight) | Janum |
| Potato (Late Blight) | Pongamia Pinnata (Healthy) |
| Soybean (Healthy) |

The data had to be manually scanned to determine whether the image was useful or not. This was necessary to train the model under the constraints of memory. Due to the original dataset being so large, the system would simply could not process the data. To optimise the process, images were omitted. The omitted images were deemed unfit for different reasons. For example, in some instances, other parts of trees were represented in the data such as tree trunks and flowers instead of the leaves, and in other cases, whole categories did not have enough images within that folder to properly train the model. It was also observable that the datasets contained some duplicates. As a result, files had to be compared to each other and the duplicated ones removed. Since the unfit images would create overheads, and with consideration for memory constraints, the focus was shifted to the images that represented a clear view of leaves which was the scope of the identification input for this system. This new dataset was split into training and testing data with 5250 images used for training, and 2500 images used in the testing process.

Modelling

This section describes the building, and training process of the model. In the early stages of the building process the team had created a model. This model only achieved an overall accuracy of around 68%. The model also had the constraint of only being able to work with 4% of the plant\_village dataset. This proved inadequate therefore the team went on to rebuild the model and create optimisations with the intention of increasing the accuracy of the model to at least 80%.

The following will be a line by line run down of how the model was built and the processes that went into training the model.

A screen shot of a computer screen

Description automatically generated with low confidence

Figure 3: showing the imported libraries to build the model

To begin the process of building the current model, the necessary libraries were imported. Majority of the libraries used were supplied by tensorflow. The other libraries were necessary for varying functionality within the building process.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 4: showing the mounting process to access the "Plant\_dataset" dataset

The aforementioned “Plant\_dataset” dataset is being mounted to the system. After the dataset has been mounted, the change directory (cd) command is used to navigate to the dataset’s directory.

A picture containing text, screenshot, software, display

Description automatically generated

Figure 5: showing how the data was split into training and testing data.

In this section of the building process, the system creates a training folder and a testing folder. The code is organizing the images in the "Plant\_dataset" directory into separate training and testing directories based on their respective classes. It moves a certain number of images from each class into the "train" directory (210 images per class) and a smaller number of images into the "test" directory (100 images per class), allowing for a train-test split in a machine learning pipeline. The line by line breakdown goes:

1. It attempts to create two directories named "train" and "test" using the os.mkdir() function. If the directories already exist, it skips this step (using the except pass block).
2. It loops through the contents of the "Plant\_dataset" directory using the os.listdir() function.
3. For each item in the "Plant\_dataset" directory, it attempts to create two directories within the "train" and "test" directories with the same name as the item. If the directories already exist, it skips this step (using the except pass block).
4. Inside each class directory, it moves the first 210 files from the "Plant\_dataset" directory to the corresponding "train" directory using the os.rename() function. This step puts 210 photos from each class into the training folder.
5. Inside each class directory, it moves the next 100 files (following the same order as before) from the "Plant\_dataset" directory to the corresponding "test" directory using the os.rename() function. This step puts 100 photos from each class into the testing folder.

A screenshot of a computer program

Description automatically generated with low confidence

Figure 6: showing the pre-processing steps and the creation of an image generator object

In this section, the model’s development begins. This code defines a function that creates an ImageDataGenerator object based on whether a pre-processing function is provided or not. The function then returns a data generator object that can be used to generate batches of image data from a specified directory during training. The line by line breakdown of the code is as follows:

1. def img\_data(dir\_path, target\_size, batch, class\_lst, preprocesss):
   1. This line defines a function named img\_data that takes several parameters: dir\_path, target\_size, batch, class\_lst, and preprocesss.
2. if preprocesss:
   1. This line checks if the preprocesss parameter is truthy (i.e., not None or equivalent to True). It is used to determine whether a preprocessing function is provided.
3. gen\_object = ImageDataGenerator(preprocessing\_function=preprocesss)
   1. This line creates an instance of the ImageDataGenerator class from the Keras library. If a preprocessing function is provided (preprocesss is truthy), it is passed as the preprocessing\_function parameter. This generator object will be used to augment and preprocess the image data during training.
4. else:
   1. This line is an else clause that executes if the preprocesss parameter is falsy (None or equivalent to False).
5. gen\_object = ImageDataGenerator()
   1. This line creates an instance of the ImageDataGenerator class without any preprocessing function specified. It will be used to generate image data without additional preprocessing.
6. return (gen\_object.flow\_from\_directory(dir\_path, target\_size=target\_size, batch\_size=batch, class\_mode='sparse', classes=class\_lst, shuffle=True))
   1. This line returns the result of the flow\_from\_directory method called on the gen\_object created in the previous steps. This method generates batches of data from the directory specified by dir\_path.
   2. target\_size is the desired size to which the images will be resized.
   3. batch\_size specifies the batch size for the generated data.
   4. class\_mode='sparse' indicates that the classes are represented as sparse integers.
   5. classes=class\_lst provides the list of class names or labels.
   6. shuffle=True specifies that the data will be shuffled between epochs during training.

A picture containing text, screenshot, font, line

Description automatically generated

Figure 7: showing the training and validation data generator objects

These lines of code set up data generators for the training and validation datasets, which will provide the necessary batches of pre-processed images for training and evaluating the model. The line by line breakdown is as follows:

1. train\_data\_gen = img\_data("train", (224,224), 500, os.listdir("train"), preprocess\_input)
   1. This line creates a data generator object for the training dataset and assigns it to the variable train\_data\_gen. The img\_data function is called with the following arguments:
   2. "train": The directory path where the training images are located.
   3. (224, 224): The target size for the images, indicating they will be resized to dimensions of 224x224 pixels.
   4. 500: The batch size for the training data, meaning the data will be divided into batches of size 500.
   5. os.listdir("train"): The list of classes or labels in the training directory.
   6. preprocess\_input: The preprocessing function to be applied to the training data.

This data generator will automatically load and preprocess the training images from the specified directory. It will generate batches of training data, where each batch will contain 500 images. The images will be resized to the specified target size and preprocessed using the preprocess\_input function.

1. valid\_data\_gen = img\_data("test",(224,224),500,os.listdir("test"),preprocess\_input)
   1. This line creates a data generator object for the validation dataset and assigns it to the variable valid\_data\_gen. The img\_data function is called with similar arguments as the previous line, but with the following differences:
   2. "test": The directory path where the validation images are located.
   3. os.listdir("test"): The list of classes or labels in the validation directory.
      1. This data generator will perform the same operations as the training data generator, but on the validation dataset. It will load and preprocess the validation images from the specified directory, generating batches of validation data with a batch size of 500.

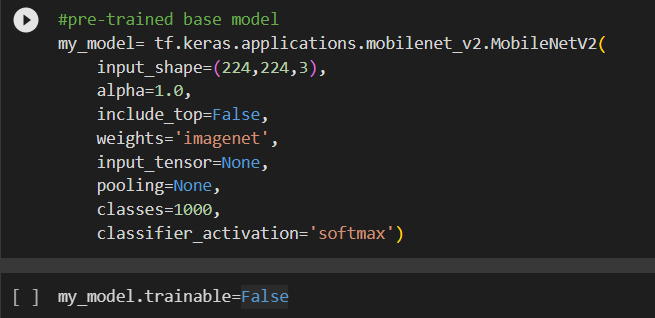


Figure 8: showing the building of the pre-trained base model

This section involves creating base model to be used as the pre-processor layer of the final model. It makes the model work more efficiently as well as increasing the accuracy of the model. A line by line breakdown of the system explains how the MobileNetV2 model’s specifications and their importance.

1. input\_shape=(224,224,3): Specifies the shape of the input images that the model will expect. In this case, it is set to (224, 224, 3), indicating images of size 224x224 pixels with 3 color channels (RGB).
2. alpha=1.0: Controls the width of the network. A value of 1.0 means that the network will have the same width as the original MobileNetV2 architecture. It can be adjusted to reduce the model size or improve computational efficiency.
3. include\_top=False: Indicates whether to include the fully connected layers at the top of the model. Setting it to False means that the final dense layers responsible for classification will not be included. This is often used when we want to customize the top layers of the model for a specific task.
4. weights='imagenet': Specifies the weight initialization for the model. By setting it to 'imagenet', the model will be initialized with pre-trained weights from the ImageNet dataset. This allows the model to already have knowledge of various visual features learned from a large dataset.
5. input\_tensor=None: Specifies the optional input tensor for the model. In this case, it is set to None, meaning that a new input tensor will be created.
6. pooling=None: Specifies the optional pooling mode for the feature extraction layer. If set to None, no pooling will be applied. Other possible values include 'avg' (average pooling) or 'max' (max pooling).
7. classes=1000: Specifies the number of classes for the classification task. This parameter is only used if include\_top=True. Since include\_top=False in this case, the number of classes is not relevant.
8. classifier\_activation='softmax': Specifies the activation function for the final classification layer. It is only relevant when include\_top=True.

By setting trainable to False, you are freezing the weights of all the layers in the my\_model object. This means that during the training process, the weights of these layers will not be updated or modified. Essentially, the model becomes a fixed feature extractor, where only the top layers (if included) can be trained. This can also help in cases where you have limited training data or computational resources. Freezing the layers reduces the number of parameters that need to be updated during training, which can lead to faster training and prevent overfitting when the training data is limited.

A screen shot of a computer

Description automatically generated with low confidence

Figure 9: showing the built CNN model.

The provided code builds a sequential model with the MobileNetV2 model as a base layer, followed by global average pooling, a fully connected layer, and an output layer. The model is then compiled with the specified optimizer, loss function, and metrics to prepare it for training.

1. model = tf.keras.models.Sequential(): Initializes a sequential model object using the Keras Sequential API. This type of model allows you to build a neural network by stacking layers sequentially.
2. model.add(my\_model): Adds the my\_model (MobileNetV2) as a base layer to the sequential model. This means that the output of the my\_model will serve as the input to the next layer in the model.
3. model.add(GlobalAvgPool2D()): Adds a global average pooling layer to the model. Global average pooling reduces the spatial dimensions of the input by taking the average value over each feature map. This helps in reducing the number of parameters and capturing important features.
4. model.add(Dense(1024, activation='relu')): Adds a fully connected layer (Dense layer) with 1024 units and ReLU activation function to the model. This layer connects every neuron from the previous layer to the 1024 neurons in this layer.
5. model.add(Dense(25, activation='softmax')): Adds the output layer to the model with 25 units (assuming there are 25 classes) and softmax activation function. Softmax activation normalizes the outputs and converts them into probability-like values, indicating the likelihood of each class.
6. model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy']): Configures the model for training. The compile() function sets the optimizer, loss function, and metrics to be used during training. In this case, the optimizer is set to Adam, the loss function is sparse categorical cross-entropy (suitable for multi-class classification), and the accuracy metric is computed to evaluate the model's performance.

A screenshot of a computer program

Description automatically generated with low confidence

Figure 10: showing the early stopping and weights training followed by the model fitting

The EarlyStopping callback stops the training process if the validation loss does not improve for a certain number of epochs, while the ModelCheckpoint callback saves the weights of the best-performing model based on the validation loss. These callbacks provide mechanisms to control the training process and save the best model for later use. The explanation of the code by line is as follows:

1. elst = callbacks.EarlyStopping(monitor='val\_loss', patience=5, mode='min'):
   1. callbacks.EarlyStopping is a callback function provided by Keras that monitors a specified metric during training and stops the training process if the metric does not improve.
   2. monitor='val\_loss' specifies that the monitored metric is the validation loss.
   3. patience=5 indicates the number of epochs to wait before stopping the training if the monitored metric does not improve.
   4. mode='min' specifies that the goal is to minimize the monitored metric (in this case, minimize the validation loss).

Overall, this callback monitors the validation loss during training and stops the training process if the validation loss does not improve for 5 consecutive epochs.

1. save\_ck = callbacks.ModelCheckpoint(".myModel.hdf5", save\_best\_only=True, monitor='val\_loss', mode='min'):
   1. callbacks.ModelCheckpoint is another callback function provided by Keras that saves the model's weights during training.
   2. ".myModel.hdf5" specifies the filename and path where the model's weights will be saved.
   3. save\_best\_only=True indicates that only the best model (based on the monitored metric) will be saved.
   4. monitor='val\_loss' specifies that the monitored metric is the validation loss.
   5. mode='min' indicates that the goal is to minimize the monitored metric (validation loss).

This callback saves the model's weights to the specified file whenever the validation loss improves compared to the previous best validation loss. This ensures that you have the best-performing model saved at the end of the training process.

During the model.fit process, the model will iterate over the training data for the specified number of epochs. For each epoch, it will perform the following steps:

* Obtain a batch of training data from the train\_data\_gen generator.
* Perform a forward pass through the network to make predictions.
* Compute the loss between the predicted output and the true labels.
* Backpropagate the gradients and update the model's weights using the optimizer.
* Repeat the above steps for each batch of training data in the epoch.
* Periodically, after processing a certain number of batches, the model will evaluate its performance on the validation data generated by valid\_data\_gen.
* The EarlyStopping callback (elst) will monitor the validation loss and stop the training process if the loss does not improve for a certain number of epochs (as specified in its parameters).
* The ModelCheckpoint callback (save\_ck) will save the model's weights to the specified file whenever the validation loss improves compared to the previous best validation loss.

These are the specifications of the model that was chosen based on the results given.

Evaluation

The model described above was chosen based on the merit of its increase in accuracy as compared to the previously built model. This model has surpassed the expected amount of 80% accuracy and has instead given an accuracy above 90%. A deeper evaluation of the model may find disparities in the accuracy percentages returned and this might be a direct result of the differences in the datasets used. The current model can be considered for early deployment with improvements expected. To show the differences in the results from each model, images of their results will be shown.

A screen shot of a computer program

Description automatically generated with low confidence A screenshot of a computer

Description automatically generated with medium confidence

Figure 11: showing the details of the first model

A screenshot of a computer program

Description automatically generated with low confidence A picture containing text, screenshot, font, menu

Description automatically generated

Figure 12: showing the details of the current model

Deployment

As can be seen, the current model presents a great improvement in accuracy and thus, it is chosen as the better model of the two.

TensorFlow Category Set:

This is a simple dictionary used to store key-value pairs. These are used to define the actual plant name.

Scalability

This system is somewhat constrained due to both hardware limitations and inaccessibility to proper expansive datasets. With access to better hardware in terms of RAM and processing power, the system will be able to train more data and therefore improve the model. With proper datasets, customised and built towards identifying plants, the system can be scaled to increase the number of plants that it can detect. Future applications of this system might find it being a module in a larger system that identifies diseases in plants based on their leaves, it may be combined with a plant based medicinal database and can be used to identify plants with medicinal benefits, and it may even be used as part of a system to identify local and indigenous plants in Jamaica, if it is fed with the right data.

# Results

Project Success

Based on the objectives defined in the Problem Definition, Herbal AI has successfully offered viable solutions. The model’s use of [methods used] has shown that the solution can successfully accept and store an image of a leaf and predict the identity of a plant based on that image. The system, however, is not without limitations which manifest in a smaller dataset, and a limited amount of RAM which constrains the model from being as expansive and exhaustive as intended by the group.

An overview of this part of the system is shown below:

A screenshot of a computer

Description automatically generated

Figure 13: showing userScreen 1 for the user interface.

Once the web application has loaded the user is able to click “Browse files”.

A screenshot of a computer

Description automatically generated

Figure 14:showing user selecting image to be identified.

A pop-up window appears giving the user the ability to navigate his/her local drive. Once the image is selected, the application imports it. The plant identification system does some processing, and the image is displayed to the user on the application screen.

The browse files button and screen achieve the group objective of the solution being able to accept images in certain file formats and this is the first step in the model solving the problem by identifying the leaf.

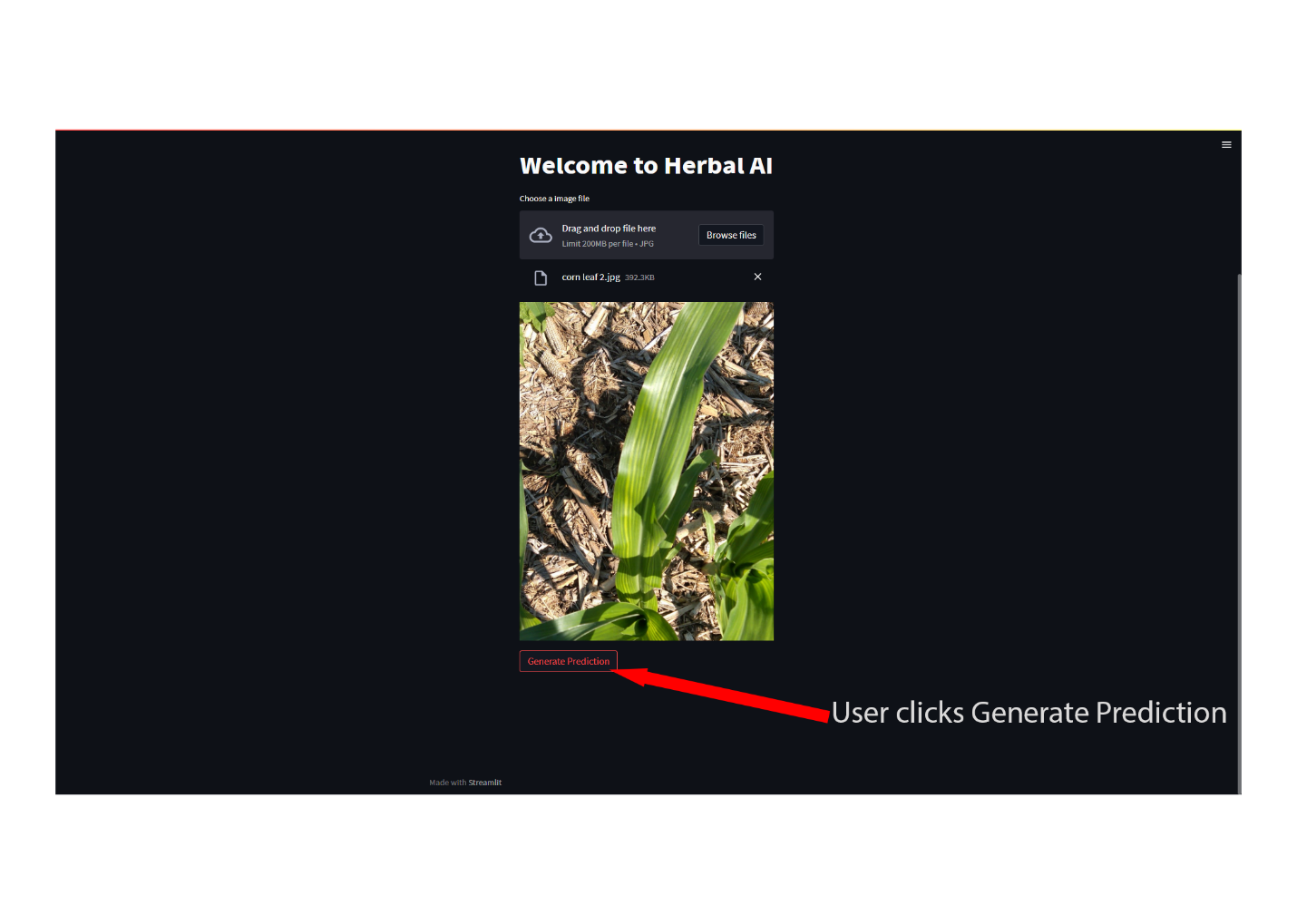


Figure 15: showing user initiating the prediction service.

A preview of the selected photo is shown on the application screen as well as a button with the text “Generate Prediction”. When this button is clicked by a user, the system passes the image through the controller component which passes it to the prediction module of the system. The module extracts data from the image and runs it though a database of the known plants. This is where the system makes a prediction based on the details extracted from the image.

The solution demonstrates its ability to temporarily store an image to facilitate the prediction being generated. This also achieves the group objective, as does the generate prediction button, which effectively kickstarts the model and returns an accurate prediction.

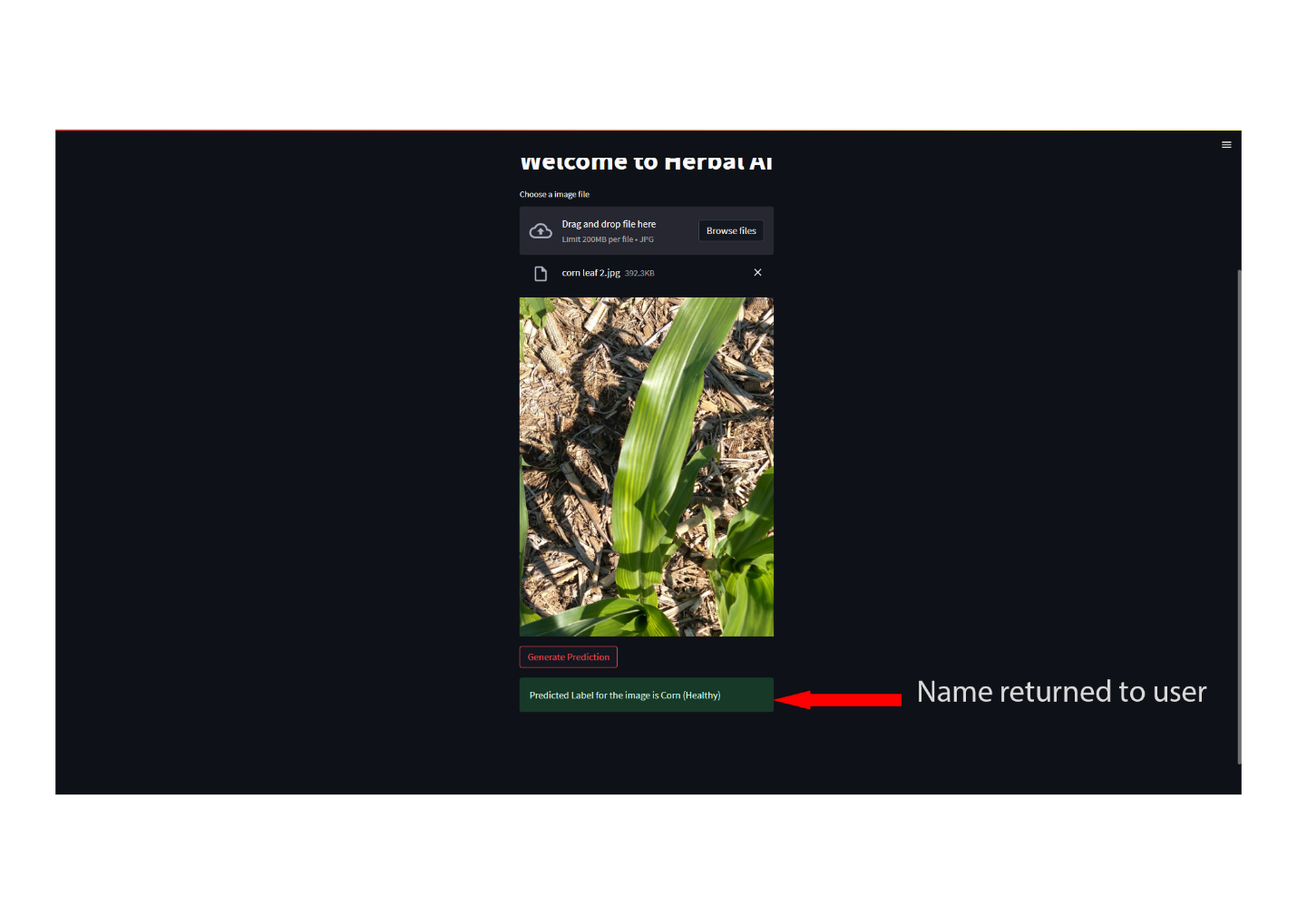


Figure 16: showing predicted plant name.

Finally, the user has a message returned with the predicted plant’s name. If the user so chooses, they may run the process again simply by clicking “Browse files” once more and beginning from step 1.

# Conclusion

Project Assessment

The final implementation of the Herbal AI solution solves the problem as defined in the problem statement above. The solution’s ability to accept images and predict the type of plant based on the intricate details of their leaves meets the objectives of the group, as well as the functional and non-functional requirements of the system as outlined in the method section.

As the solution succeeds in identifying plants with a 95% accuracy rate, it is the estimation of Herbalist Group that Herbal AI offers a comprehensive solution to the issue of plant identification and fills the gap of lightweight, accessible plant identification systems.

The applications of this system can enable botanists, agriculturalists, plant enthusiasts and casual enjoyers of plants to make various determinations based on the information provided by the software.

Alternative Approach

In assessing the completed project, there were a few alternative approaches that could have been taken to create an even better and more comprehensive solution to the problem.

**Project Scope**

The scope of our solution was to identify plants by using close images of plant leaves. The scope could have been expanded to include stems, as well as photos taken from a distance. This would increase the usability of the project, thereby solving the initial problem to a greater degree.

In addition, the scope could have been focussed on indigenous plants, or plants common to the Caribbean, which would enable the project to resonate more with regional users and differentiate it even further from similar software which may exist.

**Collaboration to Procure a Better Dataset**

The development of our project depended heavily on the availability of a suitable dataset to train and test the AI model on. The team could have collaborated with a more stakeholders like the Botany Department with an aim to gaining access to a complete and reliable dataset for the model to boast a wider array of plants and a greater number of diseases that can affect each plant. This would have aided greatly in making the project comprehensive and increasing its possible applications.

**Collaboration to Procure more Capable Hardware Resources**

A major limitation of Herbal AI in its current state is the limited dataset it employs. To remedy this situation, a larger dataset would be required to train and test the model but a complication of this is it will take a larger amount of RAM to complete, something our machines were unable to fulfil. The team could have partnered with a stakeholder with an aim of accessing a compute with greater primary storage to facilitate the greater dataset, which would bring a level of accuracy, specificity, and encyclopaedic knowledge of plant leaves to heighten the usability and value of the solution.

Lessons

Over the course of this project, the team learnt many lessons along the way which will serve to improve our future progress.

**Time Management**

The team could have made better use of time as the project schedule was not strictly adhered to, which caused issues in the implementation of the project. It is important to ensure that the time is not wasted and a timeline for a project is reasonable; the project and the sub tasks involved should be able to be completed by their assigned deadlines. It is also important to ensure the team follows a well-defined project schedule as closely as possible to ensure a project can be delivered on time without any loss of quality due to rushing to keep a deadline.

**Research**

The team could have conducted more strenuous research activity in the initial stages of the project. While our solution was well designed and adheres to a set system architecture, complications surrounding our dataset and the hardware limitations associated made our development and implementation more difficult than was necessary. It is important to conduct thorough research into all variables in both the problem definition and the proposed solution to that problem, ensuring that as many nuances are accounted for as possible.

**Collaboration**

The team has throughout the development of this project, learnt the importance of teamwork and clarity of delegation. We learnt to complete our objectives quickly and efficiently by assigning and delegating tasks to each member based on his individual strengths. This facilitated a robust project as all members invested more time into strong areas and less time on the weaker ones. We learn to work in tandem through improved communication, regular meetings and group brainstorming sessions. It is important that a team completing a project can work as a well-oiled machine to avoid and eliminate issues within project design, development, and implementation promptly and effectively.