GreenClassify: Deep Learning-based Approach for Vegetable Image Classification

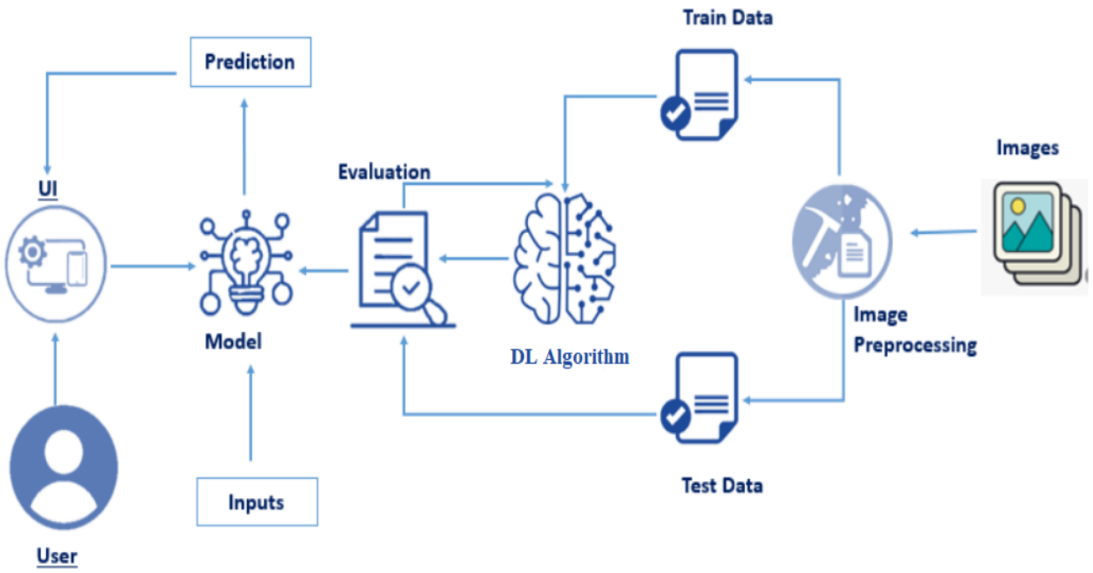
Introduction:

Vegetable image classification plays a crucial role in various domains such as agriculture, food industry, and dietary analysis. Accurate identification and classification of vegetables are essential for quality control, inventory management, and automated sorting systems. In recent years, the advent of deep learning techniques has revolutionized the field of computer vision, enabling more accurate and efficient image classification tasks.

The objective of this report is to explore and analyze the application of deep learning algorithms for vegetable image classification. By leveraging convolutional neural networks (CNNs), we aim to develop a robust and reliable system that can accurately classify different types of vegetables based on their visual attributes.

Deep learning algorithms have demonstrated remarkable success in various image recognition tasks, surpassing traditional machine learning approaches. The ability of CNNs to automatically learn hierarchical features from raw image data makes them particularly well-suited for complex classification problems. By training a CNN model on a large dataset of labeled vegetable images, we can harness its power to recognize patterns and extract discriminative features from the input data.

Technical Architecture:



Prerequisites:

To complete this project, you must require the following software's, concepts, and packages

Anaconda Navigator is a free and open-source distribution of the Python and R programming languages for data science and machine learning related applications. It can be installed on Windows, Linux, and macOS.Conda is an open-source, cross-platform, package management system. Anaconda comes with so very nice tools like JupyterLab, Jupyter Notebook, QtConsole, VScode, Glueviz, Orange, Rstudio, Visual Studio Code. For this project, we will be using collab and VS code

- Deep Learning Concepts

o CNN: a convolutional neural network is a class of deep neural networks, most commonly applied to analyzing visual imagery.

CNN Basic

o Flask: Flask is a popular Python web framework, meaning it is a third-party Python library used for developing web applications.

Project Objectives:

By the end of this project you will:

- Know fundamental concepts and techniques of Convolutional Neural Network.
- Gain a broad understanding of image data.
- Know how to pre-process/clean the data using different data preprocessing techniques.
- know how to build a web application using the Flask framework.

Project Flow:

- The user interacts with the UI (User Interface) to choose the image.
- The chosen image analyzed by the model which is integrated with flask application.
- CNN Models analyze the image, then prediction is showcased on the Flask UI.

To accomplish this, we have to complete all the activities and tasks listed below

- Data Collection: Collect or download the dataset that you want to train your CNN on.

- Data Preprocessing: Preprocess the data by resizing, normalizing, and splitting the data into training and testing sets.

- Model Building:
  a. Import the necessary libraries for building the CNN model
  b. Define the input shape of the image data
  c. Add layers to the model:
  i. Convolutional Layers: Apply filters to the input image to create feature maps
  ii. Pooling Layers: Reduce the spatial dimensions of the feature maps
  iii. Fully Connected Layers: Flatten the output of the convolutional layers and apply fully connected layers to classify the images
  d. Compile the model by specifying the optimizer, loss function, and metrics to be used during training

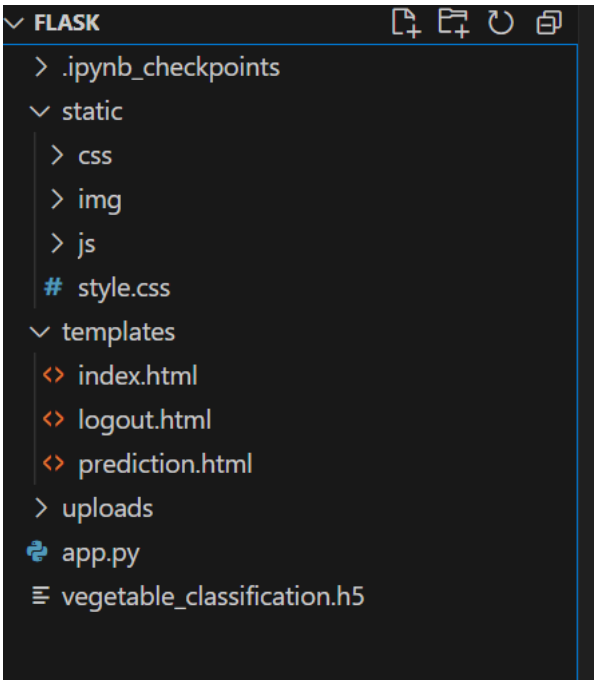
- Model Training: Train the model using the training set with the help of the ImageDataGenerator class to augment the images during training. Monitor the accuracy of the model on the validation set to avoid overfitting.

- Model Evaluation: Evaluate the performance of the trained model on the testing set. Calculate the accuracy and other metrics to assess the model's performance.

- Model Deployment: Save the model for future use and deploy it in real-world applications.

Project Structure:

Create a Project folder which contains files as shown below



Milestone 1: Data Collection

About the dataset

This dataset contains three folders:

- train (15000 images)
- test (3000 images)
- validation (3000 images)
  each of the above folders contains subfolders for different vegetables wherein the images for respective vegetables are present.

```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
```

```
! kaggle datasets download -d misrakahmed/vegetable-image-dataset
```

```
Downloading vegetable-image-dataset.zip to /content
 99% 529M/534M [00:05<00:00, 146MB/s]
100% 534M/534M [00:05<00:00, 95.6MB/s]
```

```
!unzip vegetable-image-dataset.zip
```

The provided commands are used to set up the Kaggle API in a local environment, such as your personal computer or a server. In Google Colab, you don't need to execute these commands because the Kaggle API credentials are uploaded directly to the Colab environment.

In Colab, you can upload the kaggle.json file directly by following the steps mentioned earlier: click on the folder icon on the left sidebar, click "Upload," and select the kaggle.json file from your local machine. This will upload the file to Colab, and you can proceed with using the Kaggle API without executing the !mkdir, !cp, and !chmod commands. The command !kaggle datasets download -d misrakahmed/vegetable-image-dataset is used to download the dataset named "Vegetable Image Dataset" from Kaggle. This dataset is provided by the user "misrakahmed" on Kaggle. Once executed in a code cell in Google Colab, it will initiate the download of the dataset as a ZIP file to your Colab environment.

Dataset Download link:

https://www.kaggle.com/datasets/misrakahmed/vegetable-image-dataset

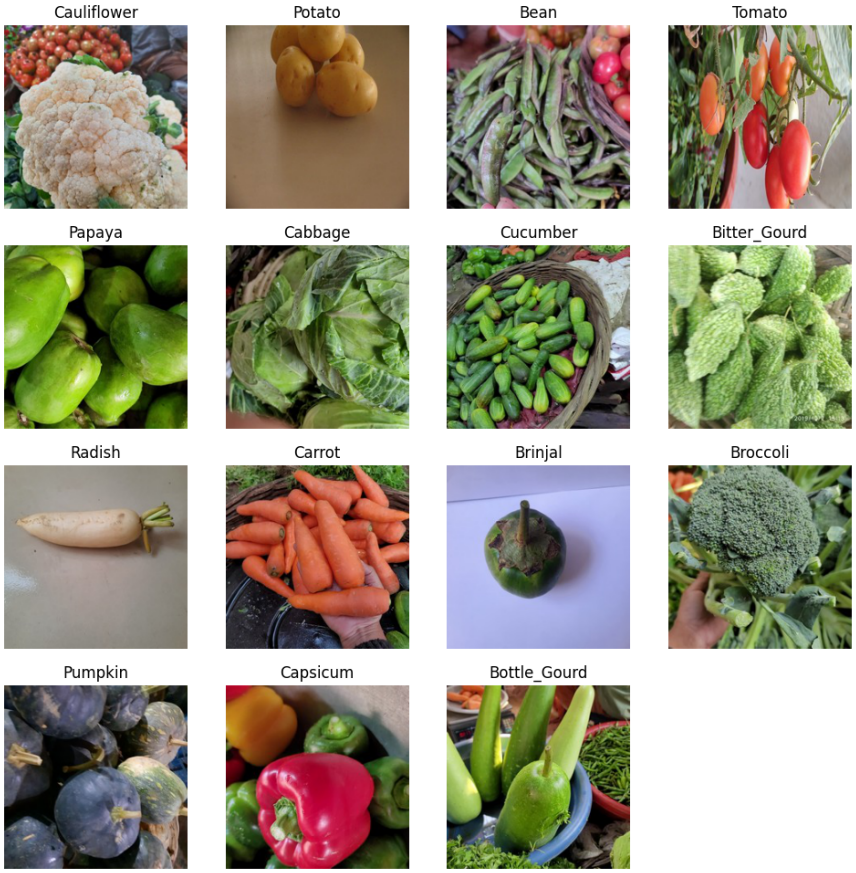
Milestone 2: Data Analysis

```
# Let's plot a few images
train_path = "/content/Vegetable Images/train"
validation_path = "/content/Vegetable Images/validation"
test_path = "/content/Vegetable Images/test"

image_categories = os.listdir('/content/Vegetable Images/train')
from keras.utils import load_img
def plot_images(image_categories):

    # Create a figure
    plt.figure(figsize=(12, 12))
    for i, cat in enumerate(image_categories):

        # Load images for the ith category
        image_path = train_path + '/' + cat
        images_in_folder = os.listdir(image_path)
        first_image_of_folder = images_in_folder[0]
        first_image_path = image_path + '/' + first_image_of_folder
        img = tf.keras.utils.load_img(first_image_path)
        img_arr =tf.keras.utils.img_to_array(img)/255.0
```



```
Found 15000 images belonging to 15 classes.
Found 3000 images belonging to 15 classes.
Found 3000 images belonging to 15 classes.
```

Milestone 3: Pre-processing

```
[ ] # Print the class encodings done by the generators
    class_map = dict([(v, k) for k, v in train_image_generator.class_indices.items()])
    print(class_map)
```

```
{0: 'Bean', 1: 'Bitter_Gourd', 2: 'Bottle_Gourd', 3: 'Brinjal', 4: 'Broccoli', 5: 'Cabbage', 6: 'Capsicum', 7: 'Carrot', 8: 'Cauliflower', 9: 'Cucumber
```

The code provided creates a dictionary class_map that maps the class indices to their corresponding labels using a generator called train_image_generator. It then prints the class_map dictionary.

Grouping Image Paths by Class Label in a Dataset

Milestone 4: Model Building

The code provided defines a sequential model in Keras. Here is a summary of the model:

- Convolutional layer with 32 filters, a kernel size of 3x3, 'same' padding, and ReLU activation.
- MaxPooling layer with a pool size of 2x2.
- Convolutional layer with 64 filters, a kernel size of 3x3, 'same' padding, and ReLU activation.
- MaxPooling layer with a pool size of 2x2.
- Flatten layer to convert the feature map into a 1D vector.
- Fully connected layer with 128 units and ReLU activation.
- Dropout layer with a dropout rate of 0.25 to prevent overfitting.
- Fully connected layer with 128 units and ReLU activation.
- Output layer with 15 units and softmax activation.
- To print the summary of the model, you can use the summary() method:

```
model = Sequential() # model object

# Add Layers
model.add(Conv2D(filters=32, kernel_size=3, strides=1, padding='same', activation='relu', input_shape=[150, 15(
model.add(MaxPooling2D(2, ))
model.add(Conv2D(filters=64, kernel_size=3, strides=1, padding='same', activation='relu'))
model.add(MaxPooling2D(2))

# Flatten the feature map
model.add(Flatten())

# Add the fully connected layers
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(128, activation='relu'))
model.add(Dense(15, activation='softmax'))

# print the model summary
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 150, 150, 32)	896
max_pooling2d (MaxPooling2D)	(None, 75, 75, 32)	0
conv2d_1 (Conv2D)	(None, 75, 75, 64)	18496
max_pooling2d_1 (MaxPooling 2D)	(None, 37, 37, 64)	0
flatten (Flatten)	(None, 87616)	0
dense (Dense)	(None, 128)	11214976
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 128)	16512
dense_2 (Dense)	(None, 15)	1935

Total params: 11,252,815
Trainable params: 11,252,815
Non-trainable params: 0

MILESTONE 5 : MODEL TRAINING

```
# Compile and fit the model
early_stopping = keras.callbacks.EarlyStopping(patience=5) # Set up callbacks
model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics='accuracy')
hist = model.fit(train_image_generator,
                 epochs=100,
                 verbose=1,
                 validation_data=val_image_generator,
                 steps_per_epoch = 15000//32,
                 validation_steps = 3000//32,
                 callbacks=early_stopping)
```

```
Epoch 1/100
```

```
Epoch 1/100
468/468 [==============================] - 66s 116ms/st
ep - loss: 1.3720 - accuracy: 0.5483 - val_loss: 0.6819
- val_accuracy: 0.7712
Epoch 2/100
468/468 [==============================] - 40s 86ms/ste
p - loss: 0.6242 - accuracy: 0.8018 - val_loss: 0.3707
- val_accuracy: 0.9005
Epoch 3/100
468/468 [==============================] - 37s 80ms/ste
p - loss: 0.3736 - accuracy: 0.8807 - val_loss: 0.2699
- val_accuracy: 0.9197
Epoch 4/100
468/468 [==============================] - 36s 76ms/ste
p - loss: 0.2525 - accuracy: 0.9186 - val_loss: 0.2946
- val_accuracy: 0.9163
Epoch 5/100
468/468 [==============================] - 40s 85ms/ste
p - loss: 0.2102 - accuracy: 0.9327 - val_loss: 0.2012
- val_accuracy: 0.9405
```

The code provided compiles and fits the model using training and validation data generators. Here is a breakdown of the code:

early_stopping: An early stopping callback is created with a patience of 5. This callback monitors the validation loss and stops training if the loss does not improve for 5 consecutive epochs.

model.compile: The model is compiled with the Adam optimizer, categorical crossentropy loss function, and accuracy metric.

hist = model.fit: The model is trained using the fit method. The parameters used are as follows:

train_image_generator: Training data generator.

epochs=100: Number of epochs to train the model.

verbose=1: Prints progress during training.

validation_data=val_image_generator: Validation data generator.

steps_per_epoch=15000//32: Number of steps (batches) per epoch, calculated based on the size of the training dataset.

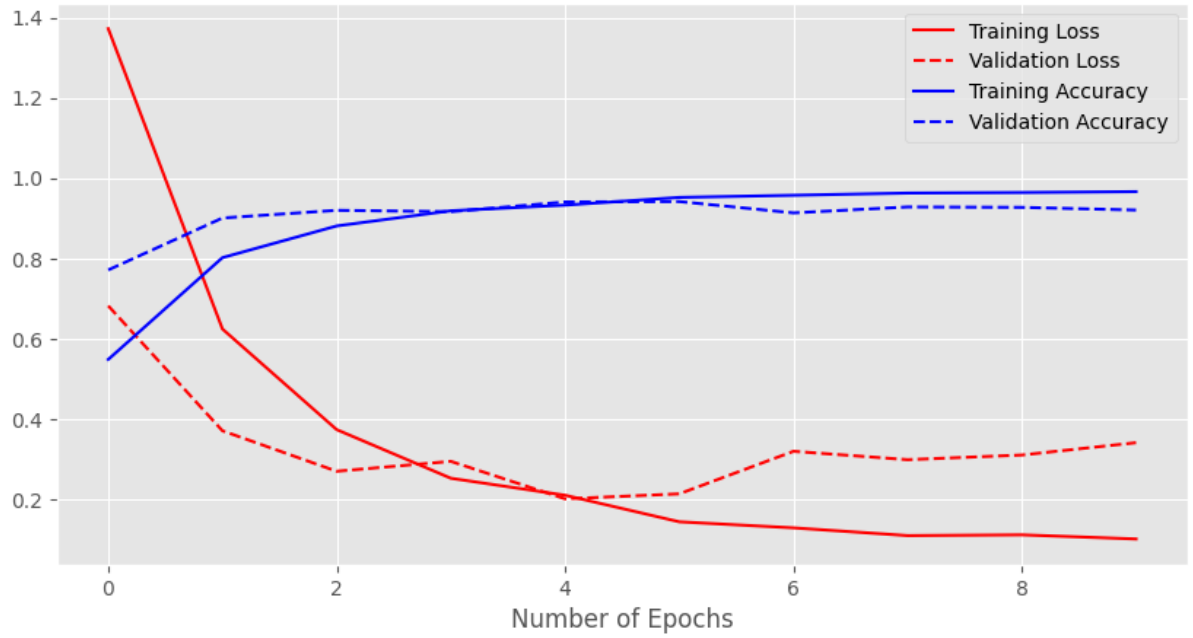
validation_steps=3000//32: Number of steps (batches) per validation epoch, calculated based on the size of the validation dataset.

callbacks=early_stopping: Early stopping callback to monitor the validation loss and stop training if there is no improvement.

The training history is stored in the hist variable.

--

```
# Plot the error and accuracy
h = hist.history
plt.style.use('ggplot')
plt.figure(figsize=(10, 5))
plt.plot(h['loss'], c='red', label='Training Loss')
plt.plot(h['val_loss'], c='red', linestyle='--', label='Validation Loss')
plt.plot(h['accuracy'], c='blue', label='Training Accuracy')
plt.plot(h['val_accuracy'], c='blue', linestyle='--', label='Validation Accuracy')
plt.xlabel("Number of Epochs")
plt.legend(loc='best')
plt.show()
```



The code provided plots the training and validation loss as well as the training and validation accuracy over the epochs. Here is a description of the code:

h = hist.history: The training history is stored in the variable h.

plt.style.use('ggplot'): Sets the plot style to "ggplot" for a visually appealing output.

plt.figure(figsize=(10, 5)): Creates a new figure with a specific size of 10x5 inches.

plt.plot(h['loss'], c='red', label='Training Loss'): Plots the training loss values over the epochs with a red color and assigns the label "Training Loss" to the line.

plt.plot(h['val_loss'], c='red', linestyle='--', label='Validation Loss'): Plots the validation loss values over the epochs with a red dashed line and assigns the label "Validation Loss" to the line.

plt.plot(h['accuracy'], c='blue', label='Training Accuracy'): Plots the training accuracy values over the epochs with a blue color and assigns the label "Training Accuracy" to the line.

plt.plot(h['val_accuracy'], c='blue', linestyle='--', label='Validation Accuracy'): Plots the validation accuracy values over the epochs with a blue dashed line and assigns the label "Validation Accuracy" to the line.

plt.xlabel("Number of Epochs"): Sets the label for the x-axis as "Number of Epochs".

plt.legend(loc='best'): Displays the legend with the labels for the lines at the best position.

plt.show(): Displays the plot.

The plot visualizes the changes in training and validation loss as well as training and validation accuracy over the epochs.

```
.4]: # Predict the accuracy for the test set
     model.evaluate(test_image_generator)

     94/94 [==============================] - 6s 63ms/step -
     loss: 0.2890 - accuracy: 0.9243

.4]: [0.28899136185646057, 0.9243333339691162]
```

```
# Predict the accuracy for the test set
model.evaluate(test_image_generator)
```

)EL EVALUATION

Load and test the model.

```python
# Testing the Model
test_image_path = '/content/Vegetable Images/test/Broccoli/1011.jpg'

def generate_predictions(test_image_path, actual_label):

    # 1. Load and preprocess the image
    test_img =tf.keras.utils.load_img(test_image_path, target_size=(150,
    test_img_arr = tf.keras.utils.img_to_array(test_img)/255.0
    test_img_input = test_img_arr.reshape((1, test_img_arr.shape[0], test

    # 2. Make Predictions
    predicted_label = np.argmax(model.predict(test_img_input))
    predicted_vegetable = class_map[predicted_label]
    plt.figure(figsize=(4, 4))
    plt.imshow(test_img_arr)
    plt.title("Predicted Label: {}, Actual Label: {}".format(predicted_ve
    plt.grid()
    plt.axis('off')
    plt.show()

# call the function
generate_predictions(test_image_path, actual_label='Brocoli')
```

Predicted Label: Bean, Actual Label: Bean

Predicted Label: Carrot, Actual Label: carrot

Predicted Label: Radish, Actual Label: radish

Predicted Label: Potato, Actual Label: potato

Milestone 6: Application Building

Now that we have trained our model, let us build our flask application which will be running in

our local browser with a user interface.

In the flask application, the input parameters are taken from the HTML page These factors are

then given to the model to know to predict the type of Garbage and showcased on the HTML

page to notify the user. Whenever the user interacts with the UI and selects the “Image” button,

the next page is opened where the user chooses the image and predicts the output.

Activity 1 : Create HTML Pages

o We use HTML to create the front end part of the web page.

o Here, we have created 3 HTML pages- index.html, prediction.html, and logout.html

o index.html displays the home page.

o prediction.html displays the prediction page

o logout.html give the result

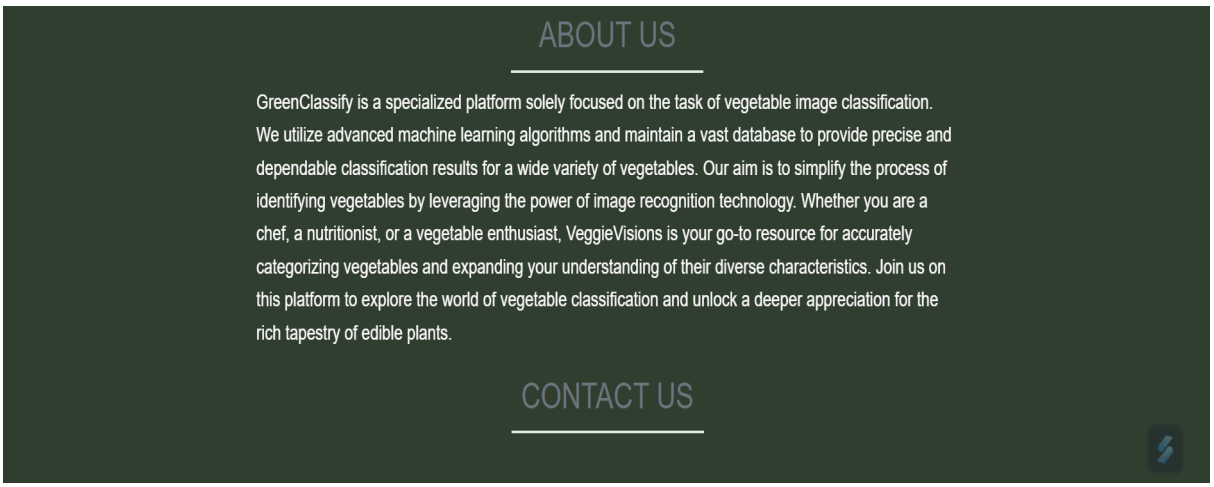
For more information regarding HTML

https://www.w3schools.com/html/

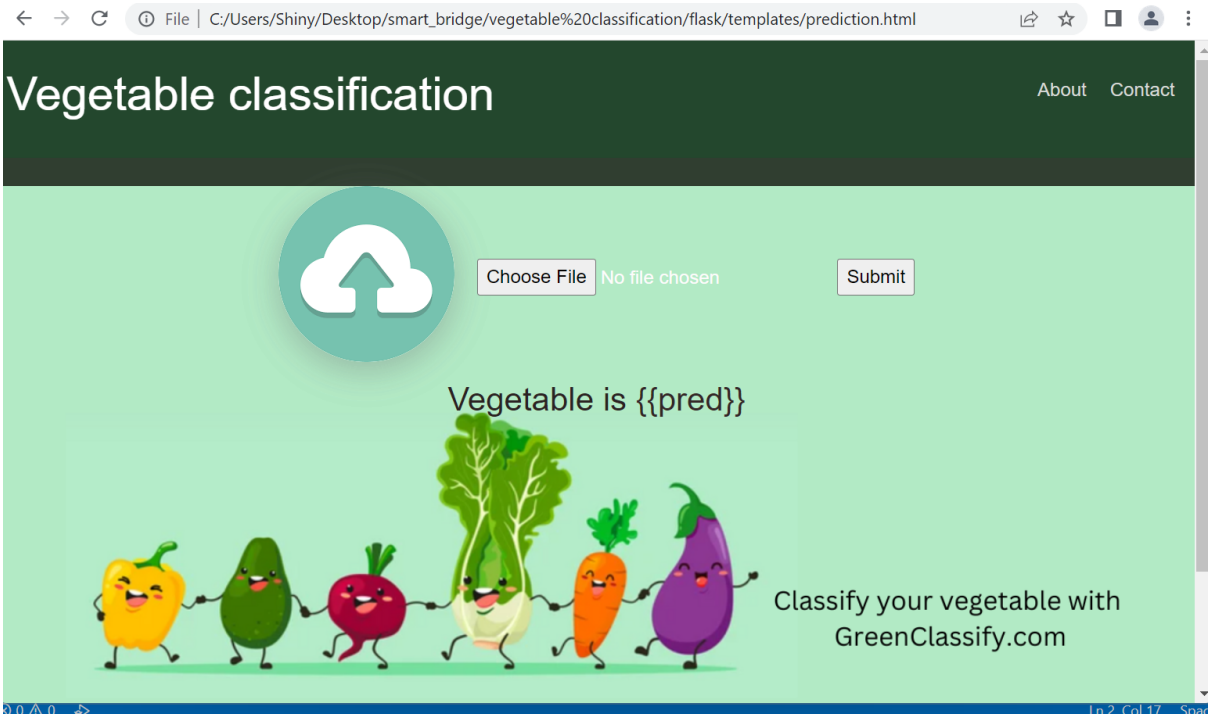
o We also use JavaScript-main.js and CSS-main.css to enhance our functionality

and view of HTML pages.

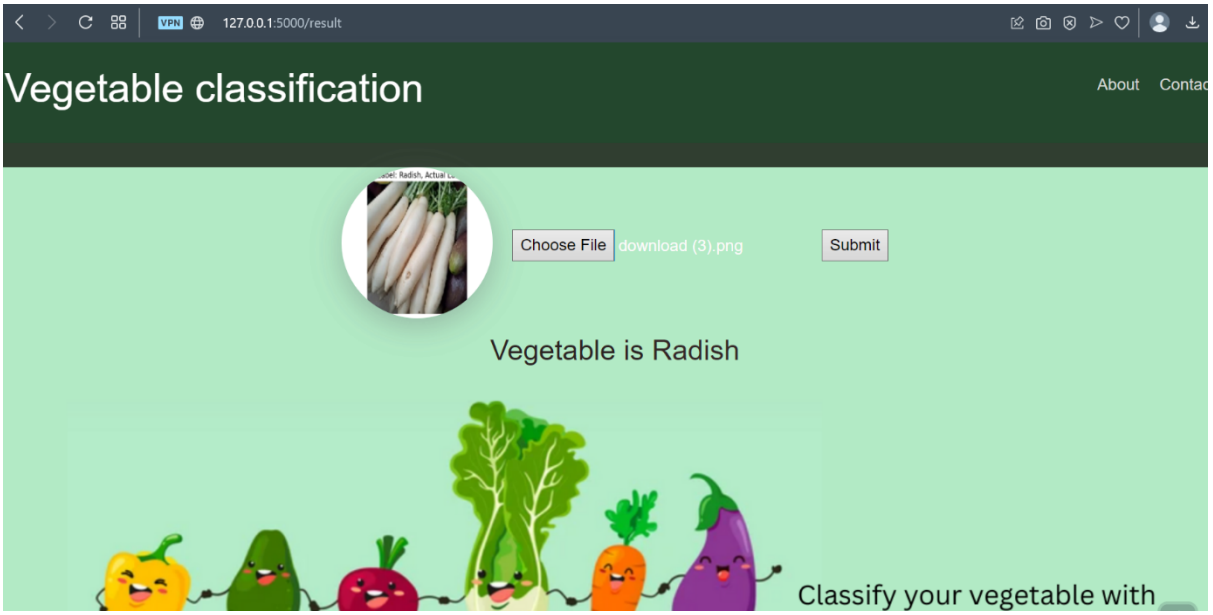
index.html :



Prediction.html:



Logout.html:



The app.py flask file :

```
import re


import numpy as np
import pandas as pd
import os
import tensorflow as tf
from flask import Flask, app,request,render_template
from keras.models import Model
from keras.preprocessing import image
from tensorflow.python.ops.gen_array_ops import Concat
from keras.models import load_model
#Loading the model
model=load_model(r"vegetable_classification.h5",compile=False)
app=Flask(__name__)

#default home page or route
@app.route('/')
def index():
    return render_template('index.html')
@app.route('/prediction.html')
def prediction():
    return render_template('prediction.html')

@app.route('/index.html')
def home():
```

```
def home():
    return render_template("index.html")

@app.route('/logout.html')
def logout():
    return render_template('logout.html')

@app.route('/result',methods=["GET","POST"])
def res():
    if request.method=="POST":
        f=request.files['image']
        basepath=os.path.dirname(__file__) #getting the current path i.e where app.py is present
        #print("current path",basepath)
        filepath=os.path.join(basepath,'uploads',f.filename) #from anywhere in the system we can give image but we want that im
        #print("upload folder is",filepath)
        f.save(filepath)
        img = tf.keras.utils.load_img(filepath,target_size=(150,150)) # Reading image
        img_arr = tf.keras.utils.img_to_array(img)

        img_input = np.expand_dims(img_arr,axis=0) # expanding Dimensions
        pred = np.argmax(model.predict(img_input)) # Predicting the higher probablity index
        op = {0: 'Bean', 1: 'Bitter_Gourd', 2: 'Bottle_Gourd', 3: 'Brinjal', 4: 'Broccoli', 5: 'Cabbage', 6: 'Capsicum', 7: 'Ca
        op[pred]
        result = op[pred]
        #result=str(op[ pred[0].tolist().op(1)])
        return render_template('prediction.html',pred=result)
```

```
        op = {0: 'Bean', 1: 'Bitter_Gourd', 2: 'Bottle_Gourd', 3: 'Brinjal', 4
        op[pred]
        result = op[pred]
        #result=str(op[ pred[0].tolist().op(1)])
        return render_template('prediction.html',pred=result)

""" Running our application """
if __name__ == "__main__":
    app.run(debug=True)
```