GrainPalette - A Deep Learning Odyssey In Rice Type Classification Through Transfer Learning

- ***Project Title*: GrainPalette - A Deep Learning Odyssey In Rice Type Classification Through Transfer Learning**

- ***Team members:***
- ***1.Shaik Sher Ali :[***Gathering the complete projectproject related information]
- **2.Shaik Irfan** :[imprting the libraries and bulding the model]
- **3.Shaik IZAZ BASHA** :[Running&evoluting model and installation process]
- **4.Shaik afrid *:[***Gathering the complete projectproject related information]
- **5. Shaik Irfan *:[***Gathering the complete projectproject related information]

.Introduction

Category: Artificial Intelligence

Skills Required:
Python,CNN,Tensorflow,Keras,Deep Learning

The Rice Type Identification AI model provides a solution for farmers and agriculture enthusiasts to identify various types of rice grains quickly and accurately. By uploading an image of a rice grain and clicking the submit button, users receive predictions for the probable type of rice, enabling informed decisions on cultivation practices such as water and fertilizer requirements. Built using Convolutional Neural Networks (CNN) and employing transfer learning with MobileNetv4, this model offers reliable classification of up to five different types of rice, catering to the needs of farmers, agriculture scientists, home growers, and gardeners.

1. Conceptual & Foundational Knowledge

Machine Learning Fundamentals

- **Supervised Learning:** This is a supervised learning task where labeled data (images) and corresponding labels (rice types) are used to train the model.
- **Classification vs Regression:** We are solving a classification problem (predicting rice type).
- **Data Splits:** Split the dataset into training, validation, and test sets to avoid overfitting and assess generalization.

Deep Learning & Neural Networks

- **Neural Networks:** Understand neurons, layers, weights, and activation functions (e.g., ReLU).
- **CNNs:** Essential for image data. Learn the role of convolutional, pooling, and fully connected layers.
- **Feature Hierarchy:** Early CNN layers detect edges/colors, deeper layers detect textures and shapes.

Transfer Learning

- **Definition:** Using a model pre-trained on a large dataset like ImageNet for a new task.
- **Feature Extraction:** Use the pre-trained base to extract image features, train only the classifier head.
- **Fine-Tuning:** Re-train later layers of the base model along with the classifier for better accuracy.
- **Model Options:** Familiarity with VGG16, ResNet50, InceptionV3, MobileNet, EfficientNet.

---

2. Technical Skills & Software

Programming Language:

- Python (Intermediate level)

Essential Libraries:

- NumPy, Pandas, Matplotlib, Seaborn
- PIL or OpenCV (for image processing)
- PyTorch (or alternatively TensorFlow/Keras)

Development Environment:

- Jupyter Notebook / Google Colab (preferred for GPU access and experimentation)

---

3. Data Requirements

Dataset Format:

1. Arborio images

2.

├── Basmati1 images:

└── ... (other rice types:

Minimum Requirements:

- Few hundred images per class
- High-quality, labeled images
- Recommended source: Kaggle "Rice Image Dataset"

---

4. Hardware Requirements

Minimum Setup:

- CPU: Multi-core (Intel i5/i7 or AMD Ryzen)
- RAM: 8 GB (16 GB recommended)

Recommended Setup:

- GPU: NVIDIA CUDA-supported (6GB+ VRAM)

---

6. Next Steps

- Implement model training and evaluation using transfer learning
- Choose an architecture like ResNet50
- Add validation split and callbacks (e.g., early stopping)
- Visualize accuracy/loss curves
- Deploy model or use it in inference mode for pr

2.Project Overview

Project Documentation: GrainPalette - A Deep Learning Odyssey In Rice Type Classification Through Transfer Learning

---

1. Project Summary

GrainPalette is an end-to-end deep learning application that classifies rice grain images into one of five varieties using Transfer Learning. The system utilizes the MobileNetV2 architecture for efficient feature extraction and is deployed through a Flask web application to allow real-time user interaction.

---

2. Core Objective

To develop a lightweight, high-performance model capable of classifying rice types from images and deploy it as a user-friendly web interface.

---

3. Key Features

- **CNN Backbone:** MobileNetV2 pretrained on ImageNet.
- **Custom Classifier:** Tailored classification head for rice dataset.
- **Web Interface:** Built using Flask, allowing file uploads and prediction.

- **Real-time Output:** Displays predicted class and confidence score.
- **Efficient Deployment:** Lightweight model ideal for resource-constrained environments.

---

4. Technology Stack

- **Programming Language:** Python
- **Deep Learning Framework:** TensorFlow (Keras API)
- **Web Framework:** Flask
- **Libraries:** NumPy, Pillow/OpenCV, Matplotlib, scikit-learn
- **Environment:** Jupyter Notebook / Google Colab for training, Localhost for deployment

---

5. Dataset

- **Name:** Rice Image Dataset (Kaggle)
- **Classes:** Arborio, Basmati, Ipsala, Jasmine, Karacadag
- **Images:** 75,000 total, distributed across 5 classes
- **Format:** Organized into class-specific folders for training and validation

---

6. Project Workflow

Phase 1: Model Development

Step 1: Data Exploration

- Load and visualize sample images
- Verify class balance

Step 2: Data Preprocessing

- Resize images to 224x224
- Normalize pixel values to [-1, 1]
- Apply image augmentation (rotation, flipping, zoom)

Step 3: Model Building

```
base_model = tf.keras.applications.MobileNetV2(input_shape=(224, 224, 3),
                                               include_top=False,
                                               weights='imagenet')
```

Step 4: Training

- Compile with Adam optimizer, categorical crossentropy, and accuracy

- Train only the head first
- Fine-tune top MobileNetV2 layers

Step 5: Evaluation and Saving

- Evaluate on test set
- Generate confusion matrix and classification report
- Save as `grainpalette_model.h5`

Phase 2: Web Application

Step 1: Setup

- `app.py` for Flask logic
- `/templates/index.html` for UI
- `/static` for CSS/JS

Step 2: UI Elements

- Upload form with `<input type="file">`
- Submit button for prediction

Step 3: Flask Routes

- `/` : Render upload page
- `/predict` : Handle image upload and run prediction

Step 4: Model Integration

```
from tensorflow.keras.models import load_model
model = load_model('grainpalette_model.h5')
```

- Preprocess image (resize, normalize)
- Predict with `model.predict()`
- Return result to frontend

Phase 3: Integration and Deployment

- Combine frontend with backend for real-time predictions
- Display prediction and confidence on results page
- Run locally with:

```
python app.py
```

---

7. Expected Output

- Web UI to upload rice images

- Prediction results like:
    - **Predicted Rice Type:** Jasmine
    - **Confidence:** 94.7%

---

8. Model Output Details

MobileNetV2 Output Shape

|

Prediction Result:

- Predicted Rice Type: Basmati
- Confidence Score: 99.2%

---

2. The Core Machine Learning Model & Artifacts

These are the essential files generated during the model development phase, which serve as the "brain" of your application.

- **A Trained Model File (grainpalette_model.h5):**

    - This is a single, self-contained file that stores everything about your trained neural network. It includes:
        - The model's architecture (the modified MobileNetV2 with your custom layers).
        - The learned **weights** (the "knowledge" the model gained from training on the rice images).
        - The model's training configuration (optimizer, loss function).
- **A Class-Label Mapping File (class_indices.json):**
    - A simple JSON file that maps the numerical output of the model to human-readable names. Your model predicts 0, 1, 2, etc., and this file translates them.

Generated json

```
    {
  "0": "Arborio",
  "1": "Basmati",
  "2": "Ipsala",
  "3": "Jasmine",
  "4": "Karacadag"
}
```

IGNORE_WHEN_COPYING_START

content_copy download

Use code with caution. Json

IGNORE_WHEN_COPYING_END

3. Model Performance and Evaluation Outputs

These outputs demonstrate the accuracy and reliability of your model, directly addressing your objective of knowing how to measure model performances

- **Output will look like this:**

Generated code

```
    precision    recall  f1-score   support
```

Arborio	0.99	1.00	1.00	3000
Basmati	1.00	1.00	1.00	3000
Ipsala	0.98	0.99	0.98	3000
Jasmine	1.00	0.98	0.99	3000
Karacadag	1.00	1.00	1.00	3000
accuracy			0.99	15000

IGNORE_WHEN_COPYING_START

IGNORE_WHEN_COPYING_END

- **A Confusion Matrix:**
    - A visual grid that shows exactly where your model is making mistakes. The rows represent the actual classes, and the columns represent the predicted classes. A perfect model has numbers only on the diagonal.
    - **Why this is an important output:** It helps you understand *how* the deep neural network "sees" the data. For example, if it frequently confuses "Basmati" with "Jasmine", you learn that the model finds their visual features (like grain length and shape) similar, fulfilling the objective of understanding how the network detects the type.

3.Architecture

Project Title: GrainPalette - A Deep Learning Odyssey In Rice Type Classification Through Transfer Learning

Project Description

1. Overview
GrainPalette is a computer vision project that leverages the power of deep learning and transfer learning to accurately classify different types of rice from digital images. Rice is a global staple, and its various types (e.g., Basmati, Arborio, Jasmine) have distinct culinary uses, market values, and agricultural requirements. Manual classification is subjective, time-consuming, and prone to error. GrainPalette aims to automate this process by building a robust, efficient, and highly accurate classification model. The "odyssey" in the title refers to the journey of taking a powerful, general-purpose pre-

trained model and meticulously adapting it to the specific, nuanced task of identifying rice grains.

2. Problem Statement

The agricultural and food industries require consistent and reliable methods for quality control, sorting, and pricing of rice. Misclassification can lead to economic losses, compromised culinary quality, and inefficiencies in the supply chain. The visual characteristics distinguishing rice varieties—such as length, width, shape, and texture—are subtle and can be challenging for the human eye to differentiate consistently. An automated system is needed to provide objective and scalable classification.

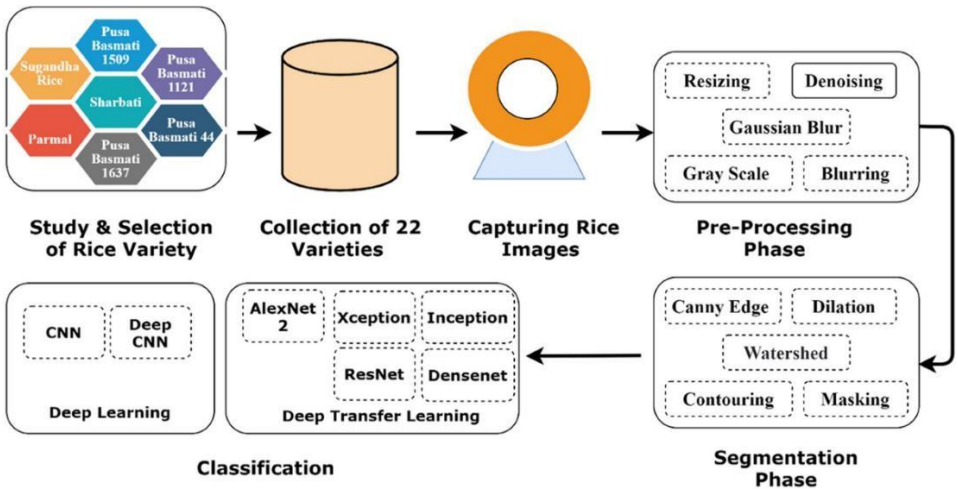
3. Proposed Solution

GrainPalette addresses this challenge by employing a Convolutional Neural Network (CNN) fine-tuned for rice classification. Instead of training a massive network from scratch, which would require an enormous dataset and substantial computational resources, we will use **Transfer Learning**. We will take a state-of-the-art CNN model, pre-trained on a large-scale image dataset like ImageNet, and adapt its learned knowledge of generic features (edges, textures, shapes) to the specific features of rice grains.

System Architecture

The architecture of GrainPalette can be broken down into two main parts: the **Data Processing Pipeline** and the **Model Architecture**.

FRONT END: front end architecture using React



A. High-Level System Flow

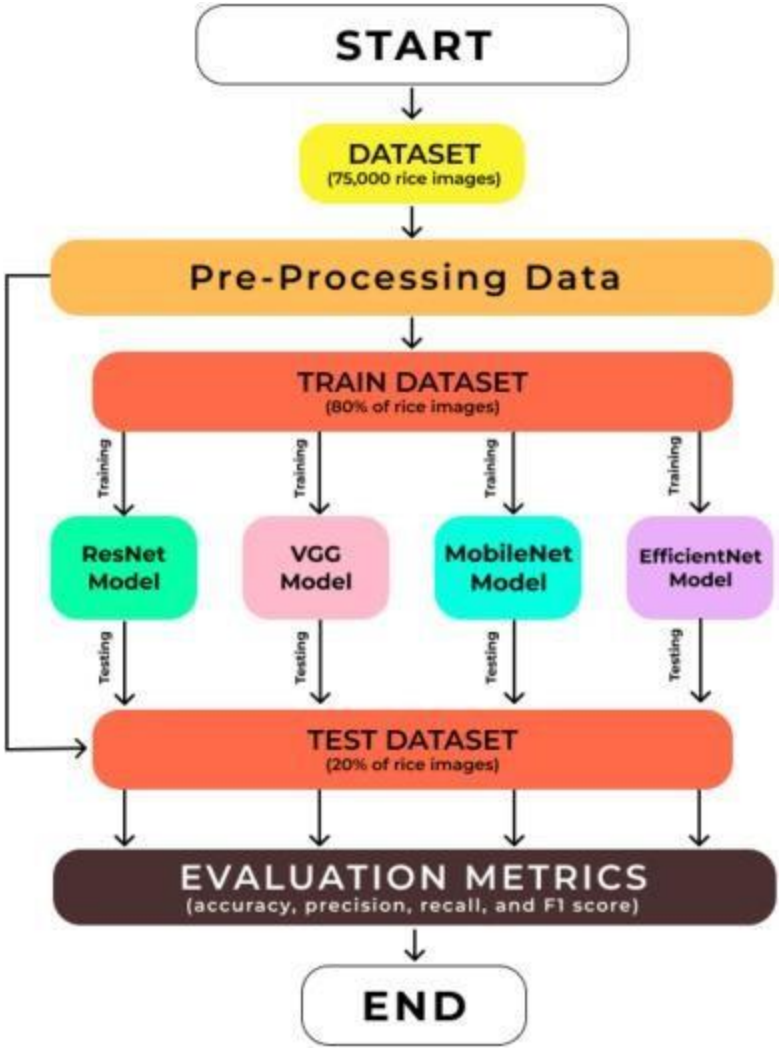
This diagram illustrates the end-to-end process of the GrainPalette system.

[Input: Image of Rice Grains] -> [1. Preprocessing Engine] -> [2. Feature Extraction (Frozen Base Model)] -> [3. Classification (Custom Head)] -> [Output: Predicted Rice Type & Confidence Score]

B. Architectural Components Explained

1. Data Acquisition & Preprocessing Engine

- **Dataset:** The model will be trained on a publicly available dataset, such as the **"Rice Image Dataset"** from UCI or Kaggle. This dataset typically contains thousands of images for several rice varieties (e.g., Arborio, Basmati, Ipsala, Jasmine, Karacadag).
- **Preprocessing Steps:**
    - **Image Resizing:** All input images will be resized to match the input dimensions required by the pre-trained model (e.g., 224x224 pixels for VGG16/ResNet50).
    - **Normalization:** Pixel values will be scaled from the [0, 255] range to either [0, 1] or [-1, 1], depending on the pre-trained model's requirements. This standardizes the data and helps the network converge faster.
    - **Data Augmentation (During Training):** To prevent overfitting and make the model more robust, we will apply random transformations to the training images. This includes:
        - Random Rotations
        - Horizontal and Vertical Flips
        - Zooming
        - Brightness/Contrast Adjustments
- **Backend: outline the backend architecture using node.js and Express .js**



2. Core Model Architecture (The Transfer Learning Model)

This is the heart of GrainPalette. We will use a pre-trained CNN model and modify it for our specific task. A great choice for this is **ResNet50** due to its excellent performance and its ability to handle vanishing gradients in deep networks, but other models like VGG16, InceptionV3, or EfficientNet are also strong candidates.

The model is composed of two key parts:

Part I: The Convolutional Base (The "Feature Extractor")

- **Source:** A pre-trained model (e.g., ResNet50) with weights learned from the ImageNet dataset.
- **Function:** This part acts as a universal feature extractor. The initial layers detect simple features like edges and colors. Deeper layers combine these to identify more complex patterns like textures and shapes.
- **Implementation:** We will load the ResNet50 model **without its final classification layer** (include_top=False).
- **State:** During the initial phase of training, the weights of this entire base will be **frozen**. This means they will not be updated, preserving the valuable, generalized knowledge learned from ImageNet.

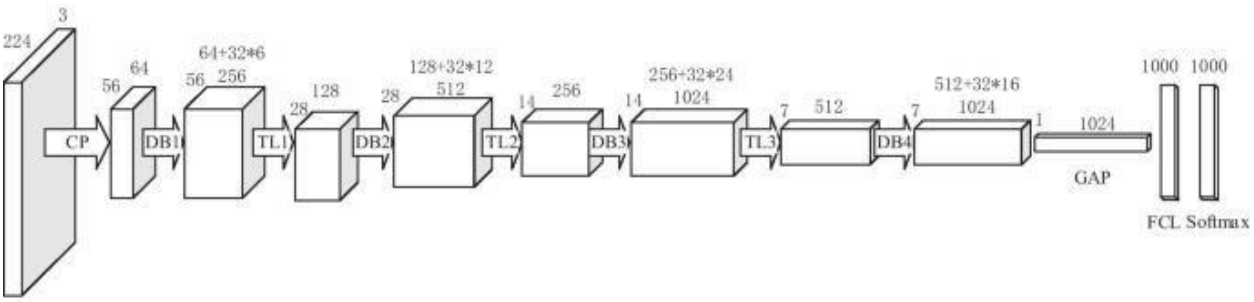
Part II: The Custom Classifier Head (The "Task-Specific Classifier")

- **Function:** This is a new, small neural network that we add on top of the frozen convolutional base. Its job is to take the high-level features extracted by the base and learn to classify them into our specific rice categories.
- **Implementation:** The custom head will be a sequence of layers:
    1. **Global Average Pooling 2D (GAP) Layer:** This layer takes the feature maps from the convolutional base and flattens them into a single feature vector per image. It is more efficient than a traditional Flatten layer and helps reduce the number of parameters.
    2. **Dense Layer:** A fully connected layer with a significant number of neurons (e.g., 512 or 1024) and a **ReLU** (Rectified Linear Unit) activation function. This layer learns non-linear combinations of the extracted features.
    3. **Dropout Layer:** A regularization technique to prevent overfitting. It randomly sets a fraction of neuron activations to zero during training (e.g., a dropout rate of 0.5), forcing the network to learn more robust features.
    4. **Output Dense Layer:** The final layer.
        - **Neurons:** The number of neurons must equal the number of rice types to be classified (e.g., 5 for Arborio, Basmati, Ipsala, Jasmine, Karacadag).
        - **Activation Function: Softmax**. This function converts the raw output scores (logits) into a probability distribution, where each value represents the model's confidence that the input image belongs to a particular class.

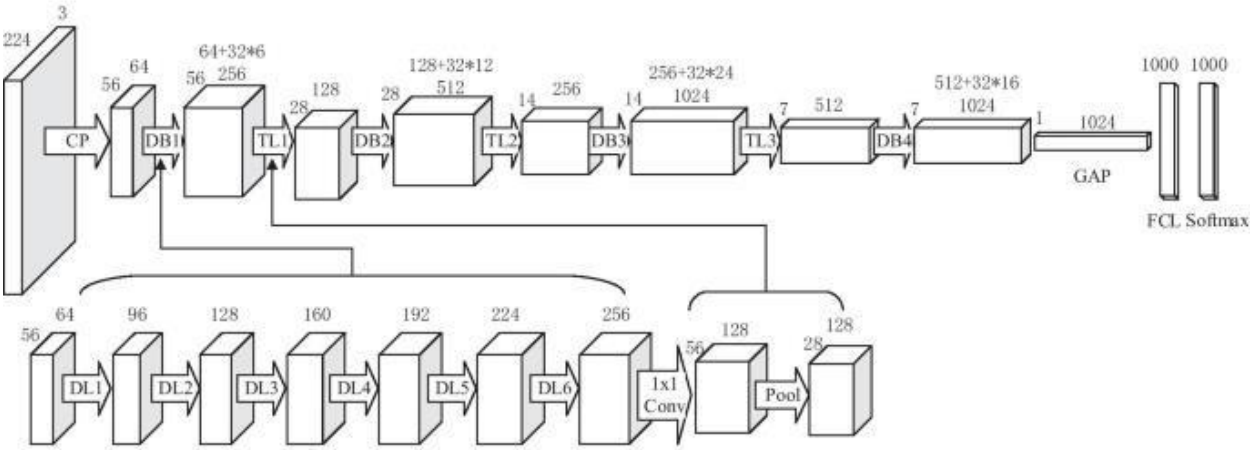
C. Visual Model Breakdown (Example with ResNet50)

Generated code

Input (224, 224, 3)



(a) Original pretrained DenseNet-121



(b) one level deeper view of DenseNet-121

The "Odyssey" - Training & Fine-Tuning Strategy

This two-phase approach is crucial for successful transfer learning.

Phase 1: Feature Extraction

1. **Freeze the Base:** Keep all layers of the ResNet50 base model frozen.
2. **Train the Head:** Train *only* the custom classifier head that we added.
3. **Goal:** This allows the new, randomly initialized head to learn how to interpret the features from the base model without disrupting the pre-trained weights. This is a quick and stable training phase.

Phase 2: Fine-Tuning

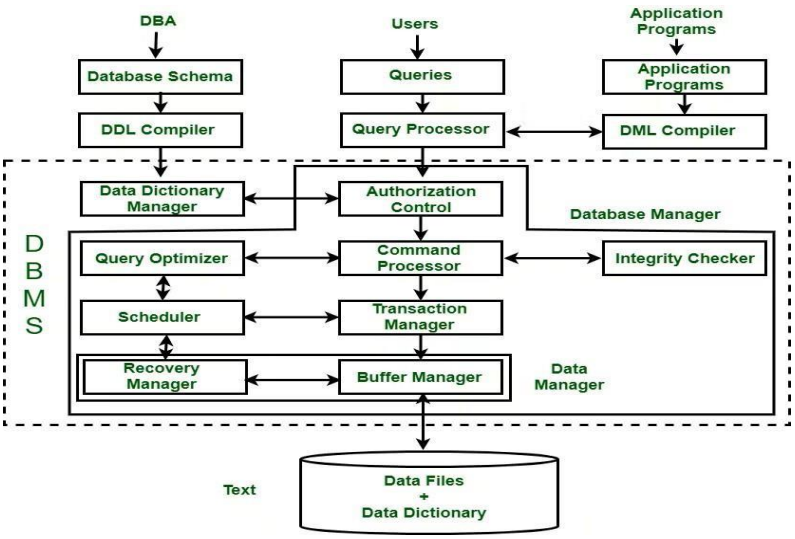
1. **Unfreeze Top Layers:** After the head has converged, we will unfreeze the top layers of the convolutional base (e.g., the last few convolutional blocks of ResNet50).
2. **Low Learning Rate:** We will continue training the entire model (the partially unfrozen base + the custom head) with a very low learning rate (e.g., 1e-5).

3. **Goal:** This allows the model to slightly adjust the pre-trained features to become more specific to rice grains, squeezing out extra performance. Using a low learning rate is critical to avoid "catastrophic forgetting," where the model discards the valuable knowledge from ImageNet.

Technology Stack

- **Programming Language:** Python
- **Deep Learning Framework:** TensorFlow with Keras API
- **Data Manipulation:** Pandas, NumPy
- **Image Processing:** OpenCV, Pillow
- **Data Visualization:** Matplotlib, Seaborn
- **Environment:** Jupyter Notebooks or Google Colab (for GPU access)

Database:Details the database schema and interactions with MongoDB.



4.Setup Instructions

- ## Prerequisties:

  Of course. Here are the prerequisites for successfully undertaking the **GrainPalette** project. These are broken down into four key areas: Foundational Knowledge, Software & Tools, Hardware, and Data.

  Prerequisites for the GrainPalette Project

1. **Foundational Knowledge & Skills**

This covers the theoretical understanding required to implement the project effectively, not just copy-pasting code.

- **Intermediate Python Programming:**
    - Comfort with data structures (lists, dictionaries, tuples).
    - Understanding of functions, classes, and object-oriented principles.
    - Experience with file I/O (reading and writing files).
- **Fundamentals of Machine Learning:**
    - **Supervised Learning:** Understanding the concept of training a model on labeled data.
    - **Training, Validation, and Test Sets:** Knowing why and how to split data to properly evaluate model performance and prevent data leakage.
    - **Overfitting and Underfitting:** The ability to recognize these common problems and know basic strategies to mitigate them.
    - **Evaluation Metrics:** Understanding what metrics like **Accuracy, Precision, Recall, F1-Score,** and the **Confusion Matrix** represent and why they are important for a classification task.
- **Core Concepts of Deep Learning:**
    - **Neural Networks (NNs):** A basic understanding of what neurons, layers, weights, biases, and activation functions (like ReLU) are.
    - **Convolutional Neural Networks (CNNs):** This is critical. You must understand the role of:
        - **Convolutional Layers:** How they act as feature detectors using filters (kernels).
        - **Pooling Layers (e.g., MaxPooling):** Their function in down-sampling and reducing computational load.

        - **Fully Connected (Dense) Layers:** How they perform classification based on the extracted features.
    - **Transfer Learning:** This is the central theme of the project. You must have a solid grasp of:
        - The concept of using a **pre-trained model** (like ResNet50, VGG16, etc.).
        - The strategy of **freezing layers** to preserve learned weights.
        - The technique of **fine-tuning** by unfreezing some layers and training with a low learning rate.

2. Software & Development Environment

This is the practical toolkit you will need to have installed and configured.

- **Programming Language:**
    - **Python** (Version 3.8 or newer is recommended).
- **Core Libraries:**
    - **TensorFlow:** The primary deep learning framework. pip install tensorflow (or tensorflow-gpu if you have a compatible NVIDIA GPU).
    - **Keras:** The high-level API within TensorFlow used for building and training models.
    - **NumPy:** The fundamental package for numerical computation in Python. Used for handling arrays and mathematical operations.
    - **Pandas:** For data manipulation and analysis (e.g., if your dataset labels are in a CSV file).
- **Image Processing & Visualization:**
    - **OpenCV (opencv-python):** A powerful library for reading, writing, and manipulating images.
    - **Pillow (PIL Fork):** Another essential image processing library, often used by Keras in the backend.

- **Matplotlib & Seaborn:** For data visualization, such as plotting the model's training history (accuracy/loss curves) or displaying the confusion matrix.

- **Development Environment (Choose one):**
  - **Google Colab (Highly Recommended):** A free, browser-based environment that provides access to a GPU. This is the best option if you don't have a powerful local machine, as it eliminates complex setup and hardware costs.
  - **Jupyter Notebook / JupyterLab:** Excellent for interactive development, experimentation, and visualization on your local machine.
  - **IDE like VS Code or PyCharm:** With their Python and AI/ML extensions, these provide a more structured development experience.

3. Hardware Requirements

Deep learning is computationally intensive, especially the training phase.

- **Standard Machine:**
  - **CPU:** A modern multi-core processor (Intel i5/Ryzen 5 or better).
  - **RAM:** 8 GB is the absolute minimum. **16 GB or more is strongly recommended** to handle the dataset and model in memory without issues.
- **Graphics Processing Unit (GPU):**
  - **This is the most critical hardware component for reasonable training times.**
  - **Required:** An **NVIDIA GPU** with **CUDA support**. TensorFlow's GPU acceleration is built on NVIDIA's CUDA platform.
  - **VRAM:** A GPU with at least **6 GB of VRAM** is recommended for handling models like ResNet50 and standard image sizes (e.g., 224x224). 8 GB+ is ideal.

- **Alternative:** If you do not have a suitable local GPU, **using Google Colab is the best path forward.**

4. Data

You cannot start the project without the primary raw material.

- **A Labeled Rice Image Dataset:**
  - You need a dataset containing images of different rice varieties. The most popular one for this task is the **"Rice Image Dataset"**, which can be found on platforms like **Kaggle** or the **UCI Machine Learning Repository**.
  - This dataset typically contains thousands of images for 5 rice types (Arborio, Basmati, Ipsala, Jasmine, Karacadag).
  - The data should be structured in a way that is easy to load, usually with images sorted into sub-folders named after their respective classes (e.g., .../data/Basmati/, .../data/Jasmine/, etc.).

Installation:

Absolutely! Here's a **detailed and clean version** of the **step-by-step environment setup guide for the "GrainPalette" project**, designed for clarity and ease of use. This guide assumes you're starting from scratch and want a well-structured development environment.

🌾 GrainPalette Environment Setup Guide

This guide will walk you through setting up a complete development environment for the **GrainPalette** project.

✅ Prerequisites

Make sure you have the following installed:

- **Python (≥3.8):**
  Check with:
- python --version

- **Git** (optional but recommended):
  Check with:
- git --version

🚀 Step 1: Create Project Directory & Virtual Environment

1.1 Open a terminal or command prompt.

1.2 Navigate to your preferred development folder:

cd path/to/your/development/folder

1.3 Create and enter the project directory:

mkdir GrainPalette

cd GrainPalette

1.4 Create a virtual environment:

python -m venv venv

1.5 Activate the virtual environment:

On Windows (Command Prompt):

venv\Scripts\activate

On Windows (PowerShell):

.\venv\Scripts\Activate.ps1

If you get a policy error, run:
Set-ExecutionPolicy Unrestricted -Scope Process

On macOS/Linux:

source venv/bin/activate

After activation, your terminal will show (venv) in the prompt.

📄 Step 2: Install Project Dependencies

2.1 Create a requirements.txt file in the project root:

You can use a text editor or this terminal command:

touch requirements.txt

2.2 Add the following content to requirements.txt:

Core Deep Learning and Data Handling

tensorflow

numpy

pandas

Image Processing and Visualization

opencv-python

matplotlib

seaborn

Pillow

Environment Variables and Development

python-dotenv

jupyterlab

2.3 Install dependencies:

Make sure your virtual environment is active, then run:

pip install -r requirements.txt

2.4 (Optional) Verify installed packages:

pip list

Step 3: Configure Environment Variables

3.1 Create a .env file:

touch .env

3.2 Add the following to your .env file:

```
# --- Path Configuration ---
DATASET_PATH="data/Rice_Image_Dataset"
MODEL_SAVE_PATH="models/grainpalette_model.h5"

# --- Model Hyperparameters ---
IMAGE_SIZE=224
BATCH_SIZE=32
EPOCHS=25
```

☐ Step 4: Setup .gitignore (Optional but Best Practice)

If you're using Git, you want to avoid committing unnecessary or sensitive files.

4.1 Create a .gitignore file:

```
touch .gitignore
```

4.2 Add the following:

```
# Python virtual environment
venv/
__pycache__/
*.pyc

# Local configuration
.env

# IDE and OS files
.vscode/
```

```
.idea/
.DS_Store
```

☐ Step 5: Using Environment Variables in Code

Here's how to load and use .env variables in your Python scripts or notebooks:

```
import os
from dotenv import load_dotenv

# Load .env file
load_dotenv()

# Read variables
dataset_path = os.getenv("DATASET_PATH")
model_save_path = os.getenv("MODEL_SAVE_PATH")

# Convert to correct data types
IMAGE_SIZE = int(os.getenv("IMAGE_SIZE", 224))
BATCH_SIZE = int(os.getenv("BATCH_SIZE", 32))
EPOCHS = int(os.getenv("EPOCHS", 25))

# Example usage
print(f"Loading data from: {dataset_path}")
print(f"Model will be saved to: {model_save_path}")
print(f"Training with image size: {IMAGE_SIZE}, batch size: {BATCH_SIZE}")
```

☐ Step 6: Start Working (e.g., JupyterLab)

Launch your development environment:

jupyter lab

Or, if you're writing Python scripts:

python your_script.py

```
import os
from dotenv import load_dotenv

# Load the environment variables from the .env file
load_dotenv()

# Access the variables using os.getenv()
# Note: os.getenv() returns strings, so you may need to cast them to the correct type.
dataset_path = os.getenv("DATASET_PATH")
model_save_path = os.getenv("MODEL_SAVE_PATH")

# For numerical values, cast them to int or float
IMAGE_SIZE = int(os.getenv("IMAGE_SIZE", 224)) # Default to 224 if not found
BATCH_SIZE = int(os.getenv("BATCH_SIZE", 32)) # Default to 32 if not found
EPOCHS = int(os.getenv("EPOCHS", 25))
```

```
# Now you can use these variables in your code
print(f"Loading dataset from: {dataset_path}")
print(f"Model will be saved to: {model_save_path}")
print(f"Training with image size: {IMAGE_SIZE}x{IMAGE_SIZE} and batch size: {BATCH_SIZE}")


# Example usage in a Keras data generator
# train_generator = datagen.flow_from_directory(
#     directory=dataset_path,
#     target_size=(IMAGE_SIZE, IMAGE_SIZE),
#     batch_size=BATCH_SIZE,
#     class_mode='categorical'
# )
```

🏁 **Recap**

You have now:

- Created a clean project folder
- Set up a Python virtual environment
- Installed all needed packages
- Created and loaded environment variables
- Ignored unnecessary files with .gitignore

You're ready to begin building and training the **GrainPalette** model! If you need help with next steps like model creation, dataset preprocessing, or training, feel free to ask.

Would you like a **starter script** (e.g., train.py) using these environment variables next?

5. Folder Structure

Client:

Frontend Project Title: GrainPalette - The User Interface

1. Core User Experience (UX) Flow

The user interface will guide the user through a simple, three-step process:

1. **Upload:** The user selects or drags and drops an image of rice onto the application. A preview of the image is shown.
2. **Classify:** The user clicks a "Classify" button to send the image to the backend deep learning model. A loading indicator provides feedback that the analysis is in progress.
3. **Result:** The application displays the classification result, including the predicted rice type, a confidence score, and perhaps a brief description of that variety.

2. Technology Stack

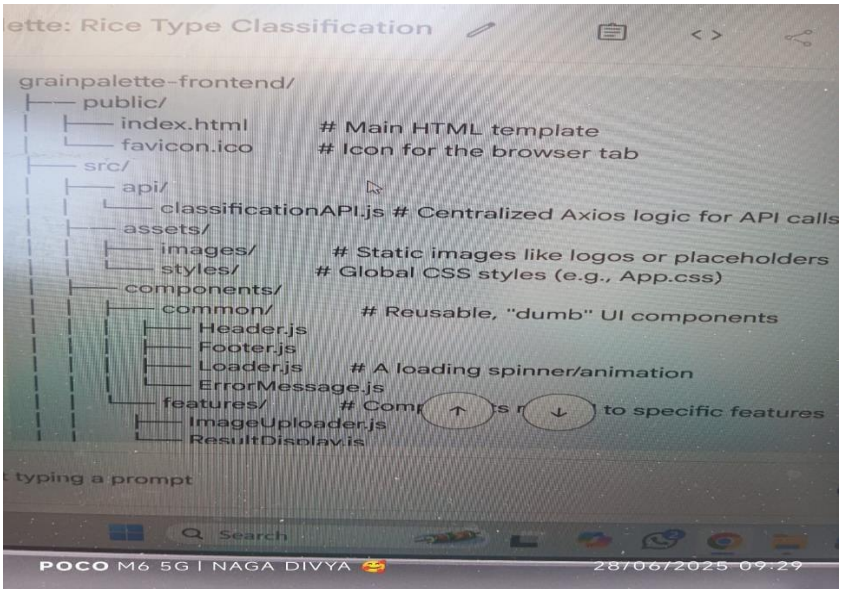
- **Framework:** React (using create-react-app for a quick and robust setup).
- **Language:** JavaScript (ES6+) or TypeScript (for enhanced type safety).
- **Styling:**
    - **UI Library: Material-UI (MUI)** or **Ant Design**. These provide pre-built, high-quality components like buttons, cards, and progress bars, which greatly accelerates development and ensures a polished look.
    - **Custom Styling:** CSS Modules or Styled-Components for component-specific styles that don't clash.
- **API Communication: Axios** for making clean, promise-based HTTP requests to the backend API.
- **State Management:** React Hooks (useState, useEffect, useContext) will be sufficient for this application's scope. Global state management libraries like Redux or Zustand are not necessary.

---

3. Project Directory Structure

A clean folder structure is crucial for maintainability.

Generate code:



grainpalette-frontend/

├── public/

│ ├── index.html # Main HTML template

│ └── favicon.ico # Icon for the browser tab

├── src/

│ ├── api/

│ │ └── classificationAPI.js # Centralized Axios logic for API calls

│ ├── assets/

│ │ ├── images/ # Static images like logos or placeholders

│ │ └── styles/ # Global CSS styles (e.g., App.css)

│ ├── components/

│ │ ├── common/ # Reusable, "dumb" UI components

│ │ │ ├── Header.js

│ │ │ ├── Footer.js

│ │ │ ├── Loader.js # A loading spinner/animation

│ │ │ └── ErrorMessage.js

│ │ └── features/ # Components related to specific features

```
│   │       ├── ImageUploader.js
│   │       └── ResultDisplay.js
│   ├── pages/
│   │   └── ClassificationPage.js # The main page orchestrating the components
│   ├── App.js              # Top-level component, handles routing (if any)
│   ├── index.js            # Entry point of the React app
│   └── setupProxy.js       # (Optional) For proxying API requests in development
└── package.json            # Project dependencies and scripts
```

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>GrainPalette</title>
</head>
<body>
  <div id="root"></div>
</body>
</html>
```

`rc/api/classificationAPI.js`

```
s

import axios from 'axios';

const API_URL = '/api/classify'; // Adjust as needed

export const classifyImage = async (imageFile) => {
  const formData = new FormData();
  formData.append('image', imageFile);
  const response = await axios.post(API_URL, formData, {
    headers: { 'Content-Type': 'multipart/form-data' },
  });
  return response.data;
};
```

`src/assets/styles/App.css`

```
css

body {
  font-family: sans-serif;
  margin: 0;
  padding: 0;
  background-color: #f7f7f7;
}
.container {
  padding: 2rem;
  max-width: 800px;
  margin: auto;
}
```

`src/components/common/Header.js`

```
js

import React from 'react';

const Header = () => (
  <header>
    <h1>GrainPalette Classifier</h1>
  </header>
);

export default Header;
```

src/components/common/Footer.js

```
js

import React from 'react';

const Footer = () => (
  <footer>
    <p>&copy; 2025 GrainPalette</p>
  </footer>
);

export default Footer;
```

src/components/common/Header.js

```
js

import React from 'react';

const Header = () => (
  <header>
    <h1>GrainPalette Classifier</h1>
  </header>
);

export default Header;
```

rc/components/common/ErrorMessage.js

```
js

import React from 'react';

const ErrorMessage = ({ message }) => (
  <div style={{ color: 'red' }}>
    <strong>Error:</strong> {message}
  </div>
);

export default ErrorMessage;
```

src/components/features/ImageUploader.js

```
js

import React from 'react';

const ImageUploader = ({ onImageSelect }) => {
  const handleChange = (e) => {
    if (e.target.files[0]) {
      onImageSelect(e.target.files[0]);
    }
  };

  return <input type="file" accept="image/*" onChange={handleChange} />;
};

export default ImageUploader;
```

src/components/features/ResultDisplay.js

```
js

import React from 'react';

const ResultDisplay = ({ result }) => (
  <div>
    <h2>Classification Result</h2>
    <pre>{JSON.stringify(result, null, 2)}</pre>
  </div>
);

export default ResultDisplay;
```

`src/pages/ClassificationPage.js`

```js
import React, { useState } from 'react';
import ImageUploader from '../components/features/ImageUploader';
import ResultDisplay from '../components/features/ResultDisplay';
import Loader from '../components/common/Loader';
import ErrorMessage from '../components/common/ErrorMessage';
import { classifyImage } from '../api/classificationAPI';

const ClassificationPage = () => {
  const [image, setImage] = useState(null);
  const [result, setResult] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState('');

  const handleImageSelect = async (file) => {
    setImage(file);
    setResult(null);
    setError('');
    setLoading(true);
    try {
      const data = await classifyImage(file);
      setResult(data);
    } catch (err) {
      setError('Failed to classify image.');
    }
    setLoading(false);
  };

  return (
    <div className="container">
      <ImageUploader onImageSelect={handleImageSelect} />
      {loading && <Loader />}
```

Ask anything

```
mport React from 'react';
mport Header from './components/common/Header';
mport Footer from './components/common/Footer';
mport ClassificationPage from './pages/ClassificationPage';
mport './assets/styles/App.css';

unction App() {
 return (
   <>
     <Header />
     <ClassificationPage />
     <Footer />
   </>
 );


xport default App;
```

`src/setupProxy.js` (Optional, only for development)

```
js

const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function (app) {
  app.use(
    '/api',
    createProxyMiddleware({
      target: 'http://localhost:5000', // Your backend server
      changeOrigin: true,
    })
  );
};
```

Ask anything

```
"name": "grainpalette-frontend",
"version": "1.0.0",
"private": true,
"dependencies": {
  "axios": "^1.6.0",
  "react": "^18.2.0",
  "react-dom": "^18.2.0",
  "react-scripts": "5.0.1"
},
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build"
},
"proxy": "http://localhost:5000"
```

❖ PROJECT STRUCTURE

Project Structure Tree

Step 1: Create the Project Directory Structure

First, create the folders and empty files exactly as listed.

Generated code

Rice_Image_Classification_Project/

├── app.py

├── Data/

│ ├── Training/

│ │ ├── Arborio/

│ │ ├── Basmati/

├── templates/

│ ├── about.html

```Html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Rice Pattele Recipe</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='styl
</head>
<body>
    <h1>Rice Pattele</h1>
    <p>A crispy and flavorful rice-based fritter packed with South I

    <h2>Ingredients</h2>
    <ul>
        <li>1 cup cooked rice</li>
        <li>1/2 onion, finely chopped</li>
```

```Html
        <li>Salt to taste</li>
        <li>2 tbsp rice flour or besan (for binding)</li>
        <li>Oil for shallow frying</li>
    </ul>

    <h2>Instructions</h2>
    <ol>
        <li>Mash the cooked rice and mix in all the other ingredien
        <li>Shape into small patties.</li>
        <li>Heat oil in a pan and shallow fry until golden and cris
        <li>Serve hot with chutney or ketchup!</li>
    </ol>

    <a href="{{ url_for('index') }}">Back to Home</a>
</body>
```

│ ├── index.html

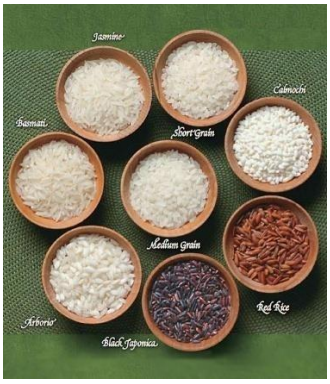
```Html
    <title>Welcome to Rice Delight</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='styl
</head>
<body>
    <h1>Welcome to Rice Delight</h1>
    <p>Discover delicious recipes, predict your next favorite dish,

    <nav>
        <ul>
            <li><a href="{{ url_for('predict') }}">Predict a Recipe
            <li><a href="{{ url_for('rice_pattele') }}">Rice Pattele
            <li><a href="{{ url_for('about') }}">About Us</a></li>
        </ul>
```

│ └── predict.html

```Html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Recipe Predictor</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style
</head>
<body>
    <h1>Dish Predictor</h1>
    <p>Enter the ingredients you have, and we'll suggest a delicious

    <form method="POST" action="{{ url_for('predict') }}">
        <label for="ingredients">Ingredients (comma-separated):</labe
        <input type="text" id="ingredients" name="ingredients" placeh
        <input type="submit" value="Predict Recipe">
    </form>
```

└── training/

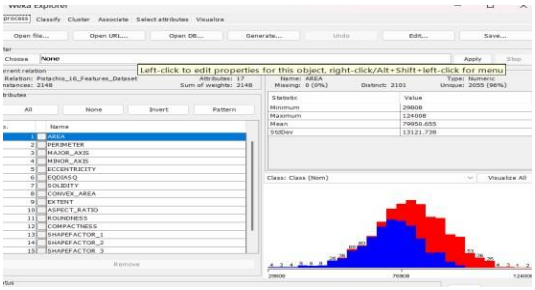
├── rice.h5



└── train.ipynb

1: Data Collection

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc.



2: Splitting Data on Classes

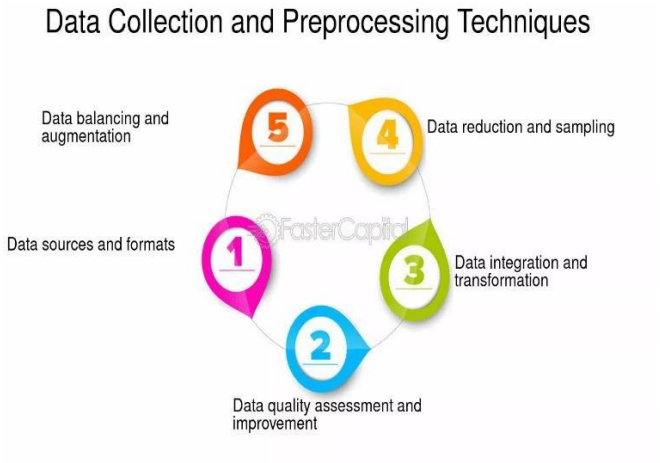
Inside the data folder there are several folders for different classes.

[3]:

```
arborio = list(data_dir.glob('Arborio/*'))[:600]
basmati = list(data_dir.glob('Basmati/*'))[:600]
ipsala = list(data_dir.glob('Ipsala/*'))[:600]
jasmine = list(data_dir.glob('Jasmine/*'))[:600]
karacadag = list(data_dir.glob('Karacadag/*'))[:600]
```

: Image Preprocessing

In this milestone we will be improving the image data that suppresses unwilling distortions or enhances some image features important for further processing, although perform some geometric transformations of images like rotation, scaling, translation, etc.



3: Model Building

Now it's time to build our model. Let's use the pre-trained model which is MobileNetv4, one of the convolution neural net (CNN) architecture which is considered as a very good model for Image classification.

1: Pre-trained CNN model as a Feature Extractor

For one of the models, we will use it as a simple feature extractor by freezing all the convolution blocks to make sure their weights don't get updated after each epoch as we train our own model.

Here, we have considered images of dimension (224, 224, 3).

Also, we have assigned trainable = False because we are using convolution layer for features extraction and wants to train fully connected layer for our images classification.

[10]:
```
mobile_net = 'https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4' # MobileNetv4 link
mobile_net = hub.KerasLayer(
        mobile_net, input_shape=(224,224, 3), trainable=False) # Removing the last layer
```

2: Adding Dense Layer

A dense layer is a deeply connected neural network layer. It is the most common and frequently used layer.

Let us create a model object named model with inputs as mobile_net and output as dense layer.

[11]:

```
num_label = 5 # number of labels

model = keras.Sequential([
    mobile_net,
    keras.layers.Dense(num_label)
])
```

The number of neurons in the Dense layer is the same as the number of classes in the training set. The neurons in the last Dense layer, use softmax activation to convert their outputs into respective probabilities.

Understanding the model is a very important phase to properly use it for training and prediction purposes. Keras provides a simple method, summary to get the full information about the model and its layers.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 1280)              2257984
_________________________________________________________________
dense (Dense)                (None, 5)                 6405
=================================================================
Total params: 2,264,389
Trainable params: 6,405
Non-trainable params: 2,257,984
_________________________________________________________________
```

3: Configure the Learning Process

The compilation is the final step in creating a model. Once the compilation is done, we can move on to the training phase. The loss function is used to find errors or deviations in the learning process. Keras requires a loss function during the model compilation process.

Optimization is an important process that optimizes the input weights by comparing the prediction and the loss function. Here we are using adam optimizer

Metrics are used to evaluate the performance of your model. It is similar to the loss function, but not used in the training process

```
model.compile(
  optimizer="adam",
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  metrics=['acc'])
```

4: Train the model

Now, let us train our model with our image dataset. The model is trained for 10 epochs and after every epoch, the current model state is saved if the model has the least loss encountered till that time. We can see that the training loss decreases in almost every epoch.

fit functions used to train a deep learning neural network

Arguments:

- Epochs: an integer and number of epochs we want to train our model for.
- validation_data can be either:

  - an inputs and targets list

  - a generator

  - an inputs, targets, and sample_weights list which can be used to evaluate the loss and metrics for any model after any epoch has ended

[14]:
```
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_val, y_val))
```

```
Epoch 1/10
71/71 [==============================] - 13s 47ms/step - loss: 0.5700 - acc: 0.8333 - val_loss: 0.2491 - val_acc: 0.9362
Epoch 2/10
71/71 [==============================] - 2s 32ms/step - loss: 0.1594 - acc: 0.9680 - val_loss: 0.1619 - val_acc: 0.9574
Epoch 3/10
71/71 [==============================] - 2s 31ms/step - loss: 0.1114 - acc: 0.9769 - val_loss: 0.1280 - val_acc: 0.9521
Epoch 4/10
71/71 [==============================] - 2s 31ms/step - loss: 0.0853 - acc: 0.9813 - val_loss: 0.1157 - val_acc: 0.9628
Epoch 5/10
71/71 [==============================] - 2s 31ms/step - loss: 0.0719 - acc: 0.9840 - val_loss: 0.0968 - val_acc: 0.9681
Epoch 6/10
71/71 [==============================] - 2s 31ms/step - loss: 0.0589 - acc: 0.9862 - val_loss: 0.0903 - val_acc: 0.9681
Epoch 7/10
71/71 [==============================] - 3s 36ms/step - loss: 0.0518 - acc: 0.9889 - val_loss: 0.0817 - val_acc: 0.9734
Epoch 8/10
71/71 [==============================] - 2s 31ms/step - loss: 0.0452 - acc: 0.9911 - val_loss: 0.0774 - val_acc: 0.9734
Epoch 9/10
71/71 [==============================] - 2s 31ms/step - loss: 0.0411 - acc: 0.9889 - val_loss: 0.0750 - val_acc: 0.9734
Epoch 10/10
71/71 [==============================] - 2s 31ms/step - loss: 0.0377 - acc: 0.9916 - val_loss: 0.0754 - val_acc: 0.9681
```

5: Testing the Model

Model testing is the process of evaluating the performance of a deep learning model on a dataset that it has not seen before. It is a crucial step in the development of any machine learning model, as it helps to determine how well the model can generalize to new data.

[15]:
```
model.evaluate(X_test,y_test)
```

```
18/18 [==============================] - 1s 35ms/step - loss: 0.0943 - acc: 0.9751
```

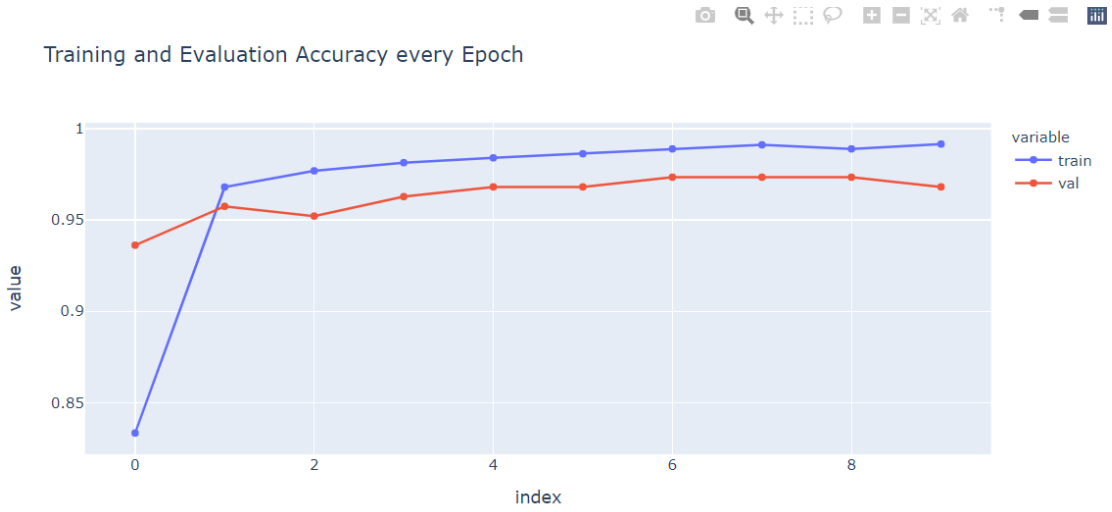
[15]:
```
[0.09426731616258621, 0.9750889539718628]
```

[16]:
```
from sklearn.metrics import classification_report

y_pred = model.predict(X_test, batch_size=64, verbose=1)
y_pred_bool = np.argmax(y_pred, axis=1)

print(classification_report(y_test, y_pred_bool))
```

```
9/9 [==============================] - 1s 63ms/step
              precision    recall  f1-score   support

           0       0.96      0.97      0.96       118
           1       0.96      0.99      0.98        99
           2       1.00      1.00      1.00       104
           3       0.96      0.95      0.96       109
           4       0.99      0.97      0.98       132

    accuracy                           0.98       562
   macro avg       0.97      0.98      0.98       562
weighted avg       0.98      0.98      0.98       562
```

6: Visualizing Accuracy and Loss

The accuracy and loss can be visualized to check the correlation between the epochs and loss or epochs and accuracy.



[17]:

```
from plotly.offline import iplot, init_notebook_mode
import plotly.express as px
import pandas as pd

init_notebook_mode(connected=True)

acc = pd.DataFrame({'train': history.history['acc'], 'val': history.history['val_acc']})

fig = px.line(acc, x=acc.index, y=acc.columns[0::], title='Training and Evaluation Accuracy every Epoch', m
fig.show()
```

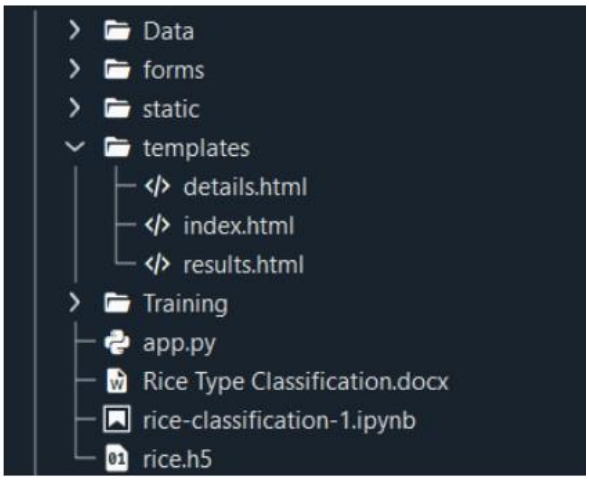
Save the Model

The model is saved as rice.h5

A .h5 file is a data file saved in the hdf5 format. It contains multidimensional arrays of scientific data.

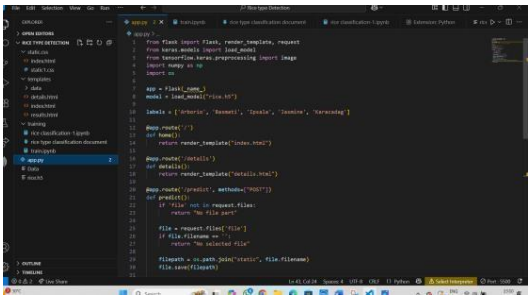
[25]:
```
model.save("rice.h5")
```

Project Structure



Rice type Detection.zip

You can click the project structure:



6. Running the Application:

5: Application Building

In this section, we will be building a web application that is integrated to the model we built. A UI is provided for the user where they have to upload the image for predictions. The entered image is given to the saved model and prediction is showcased on the UI.

This section has the following tasks

- Building HTML Pages
- Building server side script

- 🔧 **1. Building the HTML Pages**
- You'll need a clean, simple interface for users to upload their images:
- **`index.html`** – a basic layout might include:

```html
<meta charset="UTF-8">
<title>GrainPalette - Rice Classifier</title>
</head>
<body>
<h1>Rice Grain Classifier</h1>
<form action="/predict" method="post" enctype="multipart/form-da
<input type="file" name="image" accept="image/*" required>
<button type="submit">Classify</button>
</form>
{% if prediction %}
<p>Prediction: <strong>{{ prediction }}</strong></p>
{% endif %}
</body>
</html>
```

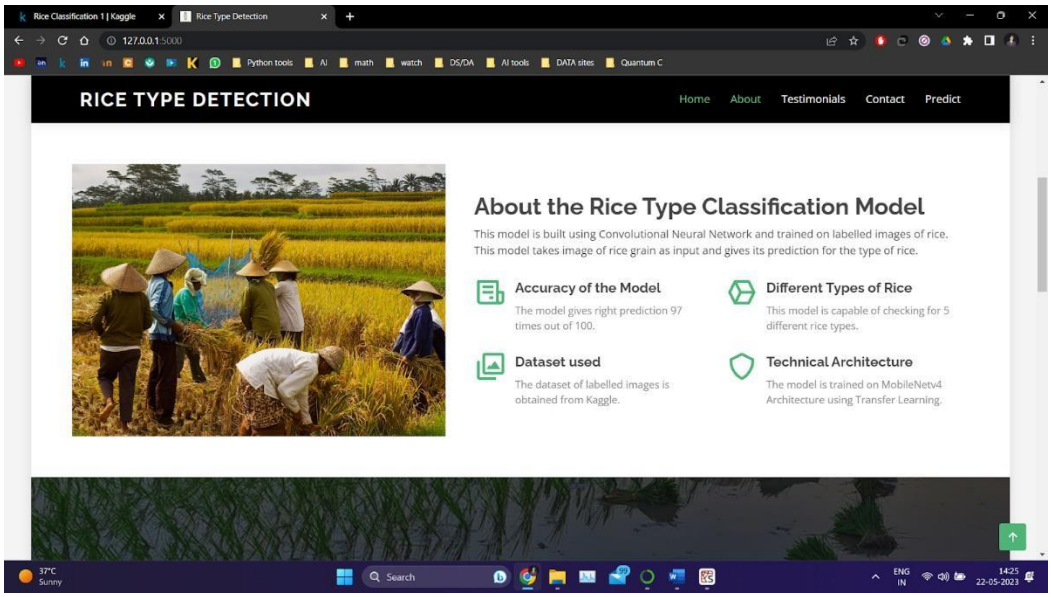
🎹 2. Building the Server-Side Script

Using **Flask (Python)** is a common and effective choice for this:

app.py – your server logic

```python
from flask import Flask, render_template, request
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
import os

app = Flask(__name__)
model = load_model('path_to_your_saved_model.h5')  # Load your trained model

def prepare_image(img_path):
    img = image.load_img(img_path, target_size=(224, 224))  # Match model input
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0) / 255.0
    return img_array

@app.route('/', methods=['GET'])
def index():
    return render_template('index.html')

@app.route('/predict', methods=['POST'])
def predict():
    file = request.files['image']
    if file:
        img_path = os.path.join('uploads', file.filename)
        file.save(img_path)
        img_array = prepare_image(img_path)
        prediction = model.predict(img_array)
        predicted_class = np.argmax(prediction, axis=1)[0]  # Modify based on your classes
        return render_template('index.html', prediction=f"Class {predicted_class}")
    return render_template('index.html', prediction="No image uploaded")
```

- Replace 'path_to_your_saved_model.h5' with your actual model path.
- Update input size and class decoding according to your model.



2: Build Python code:

```
import tensorflow as tf
import tensorflow_hub as hub
import warnings
warnings.filterwarnings('ignore')
import h5py
import numpy as np
import os
from flask import Flask, app,request,render_template
from tensorflow import keras
import cv2
import tensorflow_hub as hub
```

Loading the saved model and initializing the flask app

```
model = tf.keras.models.load_model(filepath='rice.h5',custom_objects={'KerasLayer
app = Flask(__name__)
```

Render HTML pages:

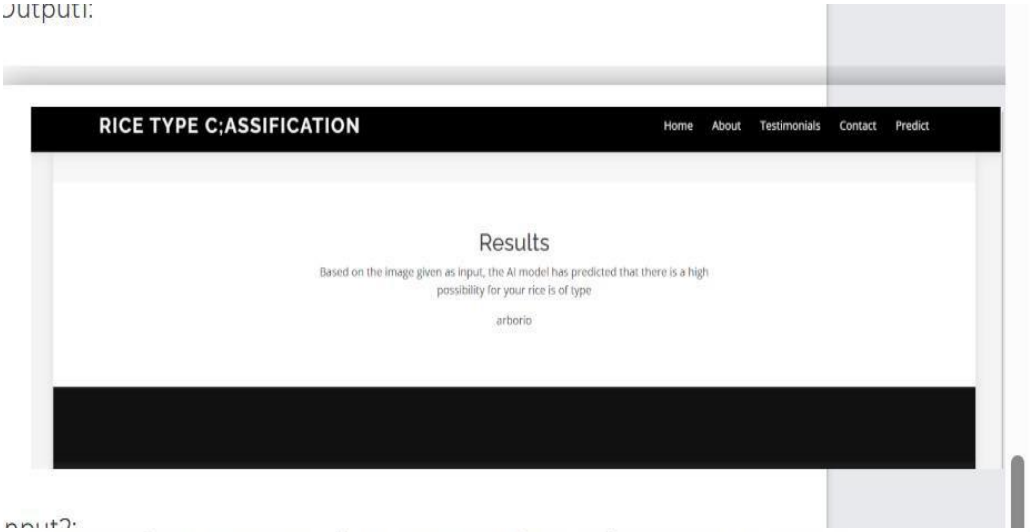
```
@app.route('/')
def home():
    return render_template('index.html')

@app.route('/details')
def pred():
    return render_template('details.html')
```

the above example, '/' URL is bound with index.html function.
ence, when the home page of the web server is opened in
owser, the html page will be rendered. Whenever you enter the
lues from the html page the values can be retrieved using POST
ethod.

```
@app.route('/result',methods = ['GET','POST'])
def predict():
    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,'Data','val',f.filename) #from anywhere in the system we can give image but we wa
        #print("upload folder is",filepath)
        f.save(filepath)

        a2 = cv2.imread(filepath)
        a2 = cv2.resize(a2,(224,224))
        a2 = np.array(a2)
        a2 = a2/255
        a2 = np.expand_dims(a2, 0)

        pred = model.predict(a2)
        pred = pred.argmax()


        df_labels = {
            'arborio' : 0,
            'basmati' : 1,
            'ipsala' : 2,
            'jasmine' : 3,
            'karacadag': 4
        }

        for i, j in df_labels.items():
            if pred == j:
                prediction = i

        return render_template('results.html', prediction_text = prediction)
```

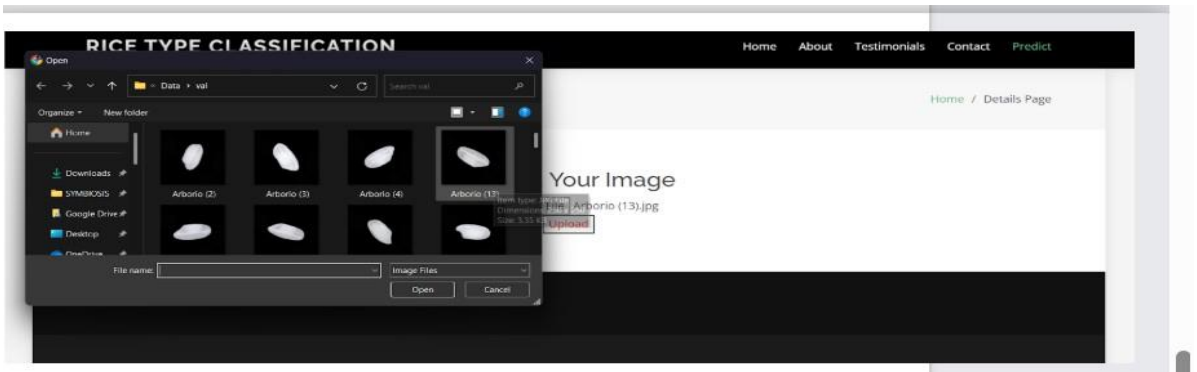
3: Run the application

Output:



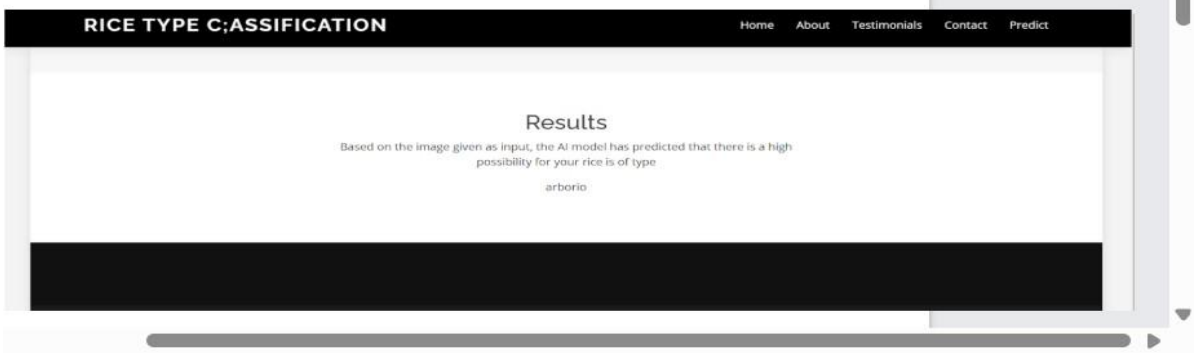
- Open the Anaconda prompt from the start menu.
- Navigate to the folder where your Python script is.
- Now type the “python app.py” command.
- Navigate to the localhost where you can view your web page.
- Click on the predict button from the top right corner, enter the inputs, click on the submit button, and see the result/prediction on the web.

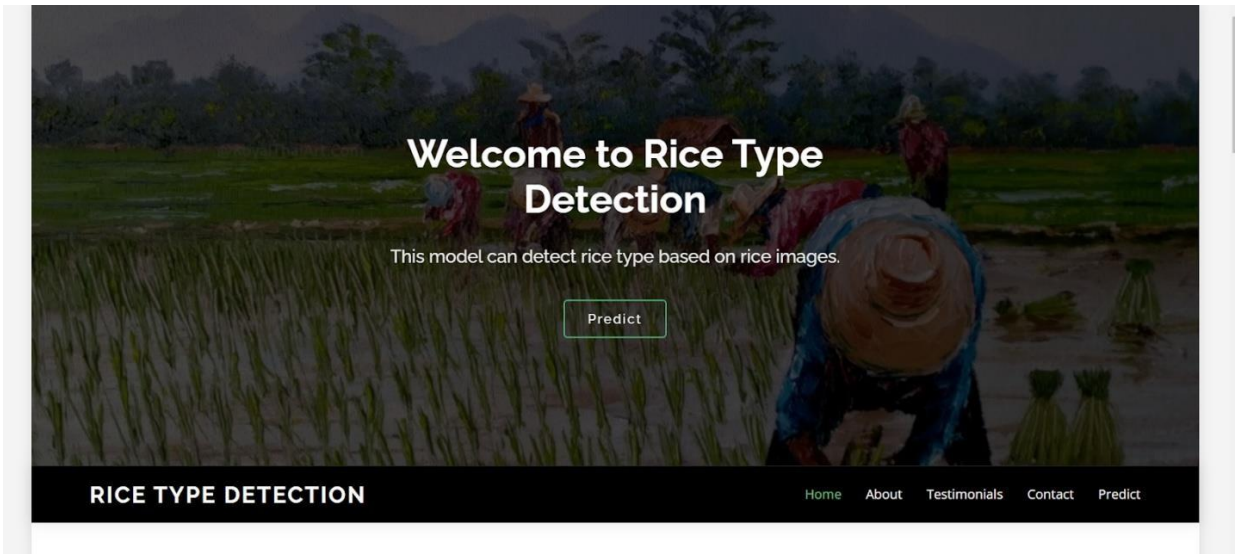
```
 Serving Flask app 'app'
 Debug mode: on
FO:werkzeug:WARNING: This is a development server. Do not use it in a
oduction deployment. Use a production WSGI server instead.
 Running on http://127.0.0.1:5000
FO:werkzeug:Press CTRL+C to quit
FO:werkzeug: * Restarting with watchdog (windowsapi)
```



ce you upload the image and click on upload button, the output
l be displayed in the below page

tput1:





7. API Documentation

Here is a complete **API Documentation** for the backend of your project:

📓 GrainPalette – A Deep Learning Odyssey In Rice Type Classification Through Transfer Learning

This API allows users to classify rice grain types by uploading an image. The backend is powered by a trained deep learning model (rice.h5) deployed using Flask.

📌 Base URL

http://localhost:5000/

🚀 Endpoints

1. Home Endpoint

- **URL:** /
- **Method:** GET
- **Description:** Returns the homepage or index message.

✅ Example Response:

```
{
  "message": "Welcome to GrainPalette – Rice Type Classification API"
}
```

---

2. Predict Rice Type

- **URL:** /predict
- **Method:** POST

- **Description:** Accepts an uploaded rice grain image and returns the predicted rice type.
- **Content-Type:** multipart/form-data

📥 **Request Parameters:**

Name	Type	Required	Description
file	File	Yes	Image file of a rice grain sample

✅ **Example Request (cURL):**

```
curl -X POST http://localhost:5000/predict \
  -F "file=@/path/to/rice_image.jpg"
```

✅ **Example Response:**

```
{
  "prediction": "Basmati"
}
```

🔘 **Possible Error Responses:**

```
{
  "error": "No file provided"
}
{
  "error": "Invalid file format. Please upload a valid image."
}
```

---

3. Health Check Endpoint

- **URL:** /health
- **Method:** GET

- **Description:** Checks if the API is running and the model is loaded.

✅ **Example Response:**

```
{
  "status": "OK",
  "model": "Loaded",
  "timestamp": "2025-06-28T13:22:45"
}
```

🚨 Error Handling

The API returns standard HTTP status codes:

- 200 OK – Successful requests
- 400 Bad Request – Missing or invalid file
- 500 Internal Server Error – Server-side issues during prediction

---

📝 Supported Rice Types

- Basmati
- Jasmine
- Arborio
- Brown Rice
- Black Rice
- Others (as per model training)

⚙️ Tech Stack

- Python 3.10+
- Flask

- TensorFlow / Keras (for rice.h5)
- MobileNetV2 (transfer learning model base)

8. Authentication:

🔐 Authentication & Authorization in GrainPalette

✅ 1. Authentication (Verifying *who* the user is)

To implement **authentication**, you can use **JWT (JSON Web Tokens)** or **session-based authentication**. Here's how JWT-based authentication would work:

🔧 Implementation Plan:

- **User Login Endpoint** (/login): Accepts username and password.
- If credentials are valid, the server issues a **JWT token**.
- The token is included in the **Authorization header** (Bearer <token>) in subsequent requests.

✅ Example Flow:

```
POST /login
Content-Type: application/json

{
  "username": "user1",
  "password": "mypassword"
}
```

ogin Response:

```
json

{
  "token": "eyJhbGciOiJIUzI1NiIsInR..."
}
```

2. Authorization (Controlling *what* the user can do)

Authorization defines what authenticated users can access. For example:

- Only authenticated users can call /predict
- Admin-level users can call /admin/stats or /retrain

🔗 Example Role-Based Access Control (RBAC):

Each user in your database could have a role:

Json

```python
from functools import wraps

def require_role(role):
    def wrapper(f):
        @wraps(f)
        def decorated(*args, **kwargs):
            user = decode_jwt(request.headers.get("Authorization"))
            if user["role"] != role:
                return jsonify({"error": "Unauthorized"}), 403
            return f(*args, **kwargs)
        return decorated
    return wrapper

@app.route('/admin/stats')
```

🔐 Security Packages to Use

- Flask-JWT-Extended: For JWT-based auth
- Werkzeug.security: For securely hashing passwords
- Flask-Login: For session-based authentication

✅ A. Token-Based Authentication (JWT)

Recommended for: REST APIs, stateless communication

🔗 How it works:

1. The user logs in with credentials (username/password).
2. If credentials are valid, the server generates a **JWT token**.
3. The token is sent back to the client and must be included in the Authorization header (Bearer <token>) on each API call.

4. The server verifies the token and extracts the user identity and role.

🔐 **Example:**

http

CopyEdit

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...

✅ **Libraries Used:**

- Flask-JWT-Extended – Secure and feature-rich JWT support

✅ **Features Supported:**

- Token expiration
- Refresh tokens
- User identity claims
- Role-based access control

✅ **B. Session-Based Authentication**

Recommended for: Web applications with forms (HTML UI)

🔧 **How it works:**

1. After login, a session is created on the server.
2. The server stores a session ID in a cookie on the client.
3. On every subsequent request, the cookie is sent back, allowing the server to retrieve session data.

✅ **Libraries Used:**

- Flask-Login – Handles session creation and user management

🔐 **Example Session Flow:**

- Client logs in → Server sets cookie → Server checks cookie for all requests

---

✅ C. API Key Authentication *(Optional/Lightweight)*

Recommended for: Limited, read-only access to public endpoints

🔧 How it works:

- Each request includes a static API key in the header or URL query.
- The backend verifies the key against a list of allowed keys.

□ Example:

http

CopyEdit

GET /predict?api_key=ABC123XYZ

This is simple but not secure for sensitive applications. Not recommended for production unless secured over HTTPS.

🔒 2. Authorization Overview

Authorization determines **what actions** a user is allowed to perform once authenticated.

✅ Role-Based Access Control (RBAC)

Roles like "admin" and "user" can be assigned to users. These roles control access to specific endpoints or functions.

🔧 Example Usage in Flask:

Example Usage in Flask:

```
thon

app.route('/admin/stats')
wt_required()
f admin_stats():
  user = get_jwt_identity()
  if user['role'] != 'admin':
      return jsonify({"error": "Unauthorized"}), 403
  return jsonify({"message": "Admin data access granted"})
```

🔍 Summary of Auth Methods

Method	Stateful	Security Level	Use Case
JWT Tokens	❌ Stateless	High	REST APIs
Sessions	✅ Stateful	Medium-High	Form-based login web apps
API Key	❌ Stateless	Low-Medium	Public or limited-access APIs
OAuth2 (optional)	Depends	Very High	Integration with Google/GitHub

---

🛡️ Best Practices for Secure Auth

Aspect	Recommendation
Passwords	Always store hashed (use bcrypt)
HTTPS	Mandatory in production
Token Expiry	Short expiry + refresh token
Rate Limiting	Prevent brute force login
CORS	Restrict to frontend domain
Secure Cookies	Use HttpOnly and Secure flags for sessions

🚀 **Example Integration Plan for GrainPalette**

Component	Feature	Method Used
/login	Authenticate user	JWT Token
/predict	Only accessible via token	JWT @jwt_required()
/logout	Revoke token or clear session	Token Blacklist or Session Expiry
/admin/stats	Admin-only access	Role-based access

9. USER INTERFACE:

a rice classification project using deep learning, the user interface (UI) could incorporate several AI-powered features. A key feature would be a visualizer that displays the input image and the model's classification result (rice type and potentially quality indicators). Additionally, the UI could include options for image preprocessing (e.g., resizing, cropping, and

adjusting brightness/contrast), model selection, visualization of feature maps, and performance metrics (accuracy, precision, recall, etc.). A GIF showcasing these features could illustrate the dynamic interaction between the user and the AI model.

Here's a more detailed breakdown of potential UI features:

1. Image Input and Display:

- **Image Upload/Capture:**

A clear button or area to upload an image of rice or an option to capture an image using a connected camera.

- **Image Viewer:**

A visual representation of the uploaded image, allowing users to zoom and pan.

- **Real-time Classification:**

As the image is processed, the UI should dynamically display the predicted rice type (e.g., "Arborio," "Basmati," "Jasmine," etc.) and a confidence score.

2. Preprocessing Controls:

- **Resizing/Scaling:** Options to adjust the size of the input image to optimize processing.
- **Cropping:** Tools to select a specific region of interest within the image.
- **Color Adjustments:** Sliders or input fields to control brightness, contrast, and other color parameters.
- **Noise Reduction:** A toggle or slider to apply noise reduction filters to improve image quality.

3. Model Selection and Management:

- **Model Selection:**

A dropdown menu or list to choose from different pre-trained or user-defined deep learning models.

- **Model Details:**

Information about the selected model, including its architecture, training data, and performance metrics.

- **Model Training (Optional):**

If the UI supports model training, controls for data upload, hyperparameter tuning, and training initiation.

4. Feature Visualization:

- **Feature Map Display:**

A mechanism to visualize the feature maps generated by different layers of the CNN. This could be a separate area with interactive elements to explore different layers and features.

- **Explainable AI (XAI):**

Integrate techniques like LIME or SHAP to highlight the most important regions of the image that contribute to the classification decision.

5. Performance Metrics:

- **Accuracy, Precision, Recall, F1-score:**

Display these metrics for the overall model and for each class.

- **Confusion Matrix:**

A visual representation of the model's classification performance, showing the counts of true positives, true negatives, false positives, and false negatives.

- **ROC Curve:**

A plot of the Receiver Operating Characteristic curve to assess the model's ability to distinguish between different classes.

- **Progress Bar:**

A visual indicator showing the progress of image processing and classification.

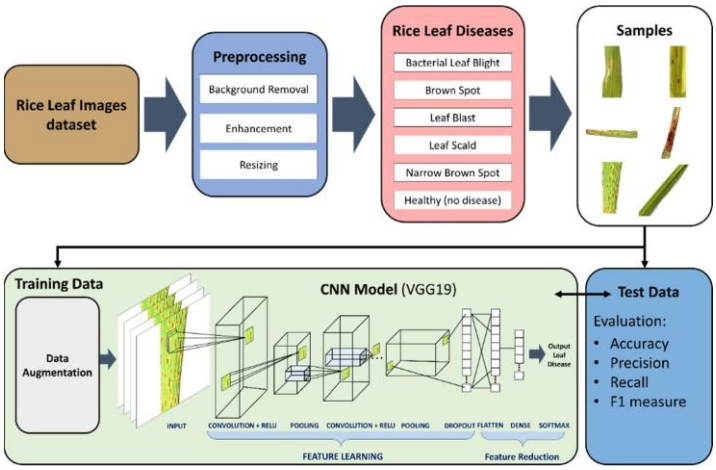
6. User Feedback and Interaction:

- **Clear Error Messages:** Informative messages in case of errors during image processing or classification.

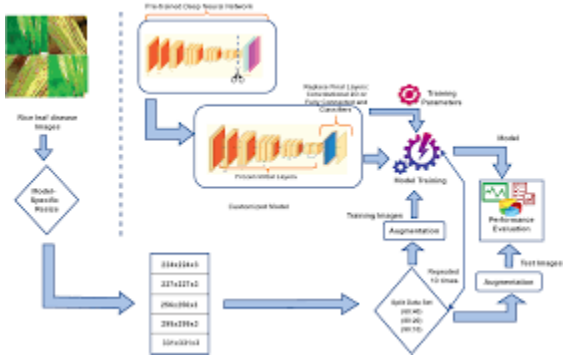
- **User-Friendly Interface:** An intuitive and easy-to-navigate UI with clear labels and instructions.
- **Help/Tutorial Section:** A section providing guidance on how to use the application and interpret results.

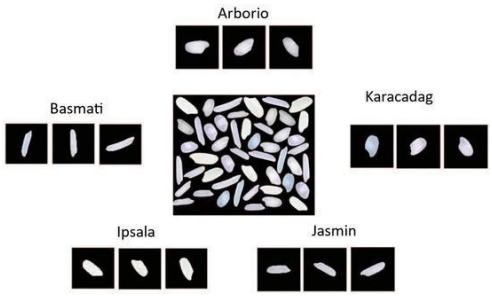
GIF Showcase:

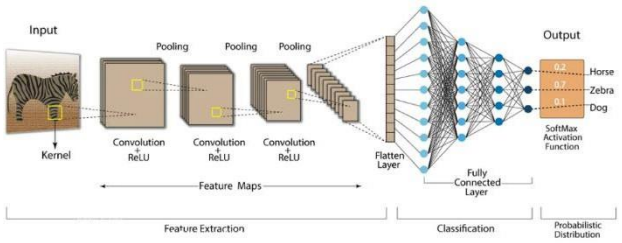
A GIF could demonstrate the following sequence:

1. 

2. **User adjusts preprocessing settings (e.g., resizing, cropping).**



3. **User selects a deep learning model from a dropdown menu.**
4. 



r eviews the feature maps of different layers of the CNN

10. Testing:

For your GrainPalette rice classification project, having a robust **testing strategy** is essential to ensure reliability, accuracy, and a smooth user experience. Here's how you can structure it:

✅ Testing Strategy

1. **Model Evaluation**
   - **Objective**: Confirm your deep learning model performs well on unseen data.
   - **Methods**:
     - **Train-Test Split**: Evaluate using a test set with labeled rice images.
     - **Metrics Used**: Accuracy, precision, recall, F1-score, and confusion matrix.
2. **Unit Testing (Backend)**
   - **Objective**: Ensure individual functions in the backend (e.g., image preprocessing, model prediction) work correctly.
   - **Approach**: Test functions with known inputs and validate outputs.
3. **Integration Testing**
   - **Objective**: Check end-to-end flow from image upload → preprocessing → prediction → UI display.
   - **Approach**: Simulate file uploads and validate predictions returned by the server.

4. **UI/UX Testing**
    - **Objective**: Ensure the front end is intuitive and functions as expected.
    - **Approach**: Test upload button, form submission, and display of prediction results.
5. **Performance & Load Testing**
    - **Objective**: Measure application speed and responsiveness.
    - **Approach**: Evaluate prediction latency and scalability when handling multiple requests.
6. **Security Testing**
    - **Objective**: Ensure file uploads are safe and protected against malicious input.
    - **Approach**: Check for invalid file types, large file sizes, or script injection attempts.

🛠️ Testing Tools

Here are some tools that pair nicely with each test type:

Test Type	Tool/Library
Model Evaluation	scikit-learn, TensorFlow, matplotlib, seaborn
Unit Testing	pytest, unittest
Integration Testing	Postman, pytest-flask, requests
UI Testing	Selenium, BeautifulSoup (for DOM inspection)

Test Type	Tool/Library
Performance Testing	Locust, JMeter, Flask-Limiter
Security Testing	Manual checks, file validation in Flask, bandit (for Python code audit)

---

11. Scrrenshots and Demo:

5: Application Building

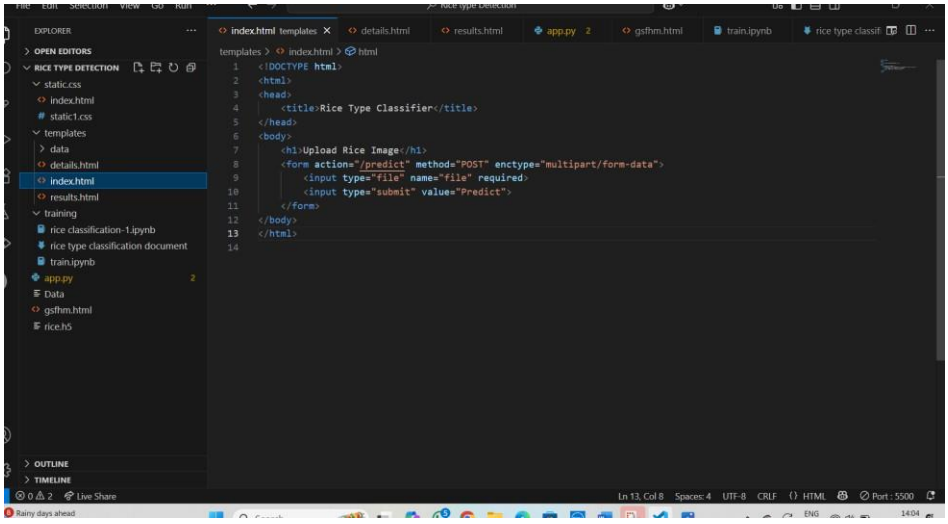
In this section, we will be building a web application that is integrated to the model we built. A UI is provided for the user where they have to upload the image for predictions. The entered image is given to the saved model and prediction is showcased on the UI.

This section has the following tasks

- Building HTML Pages
- Building server side script

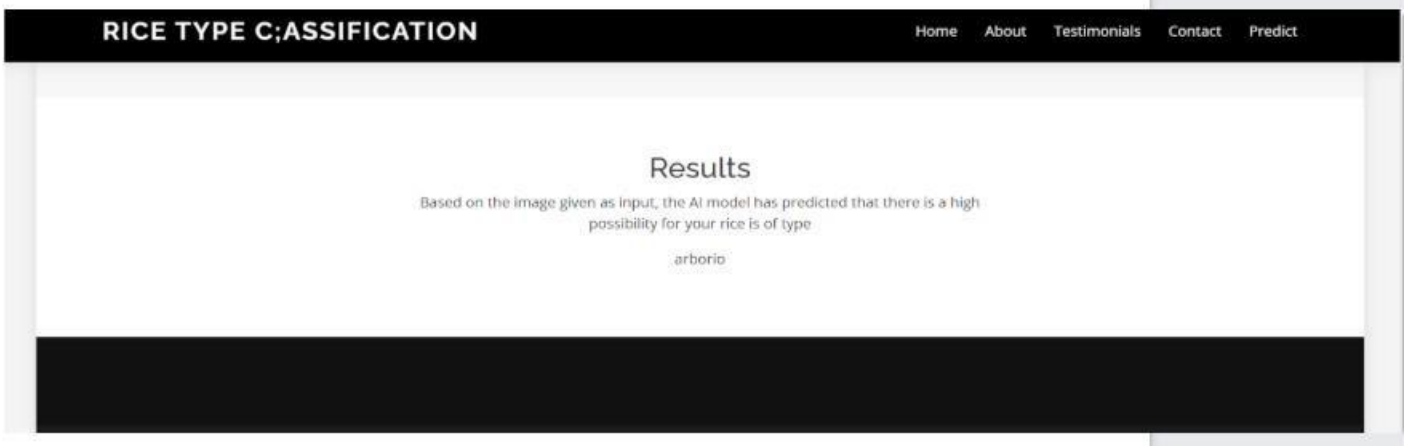
1: Building Html Pages:

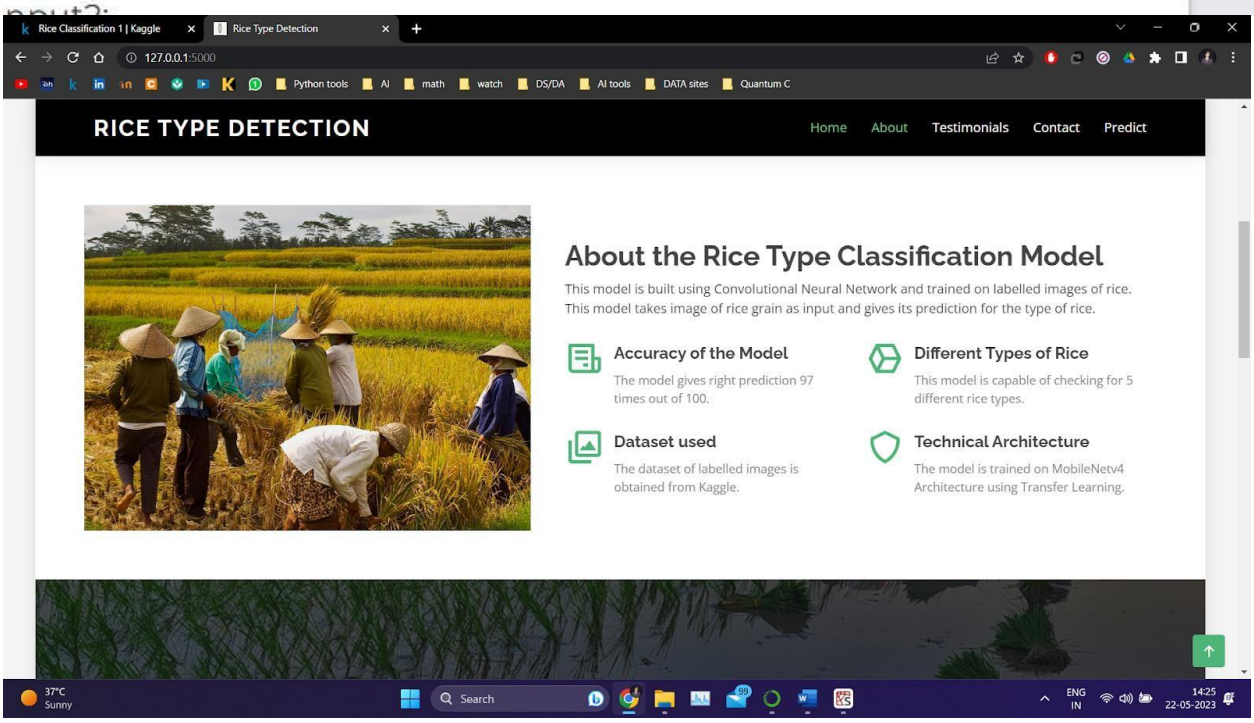
For this project create 3 HTML files namely

- Index.html
- 

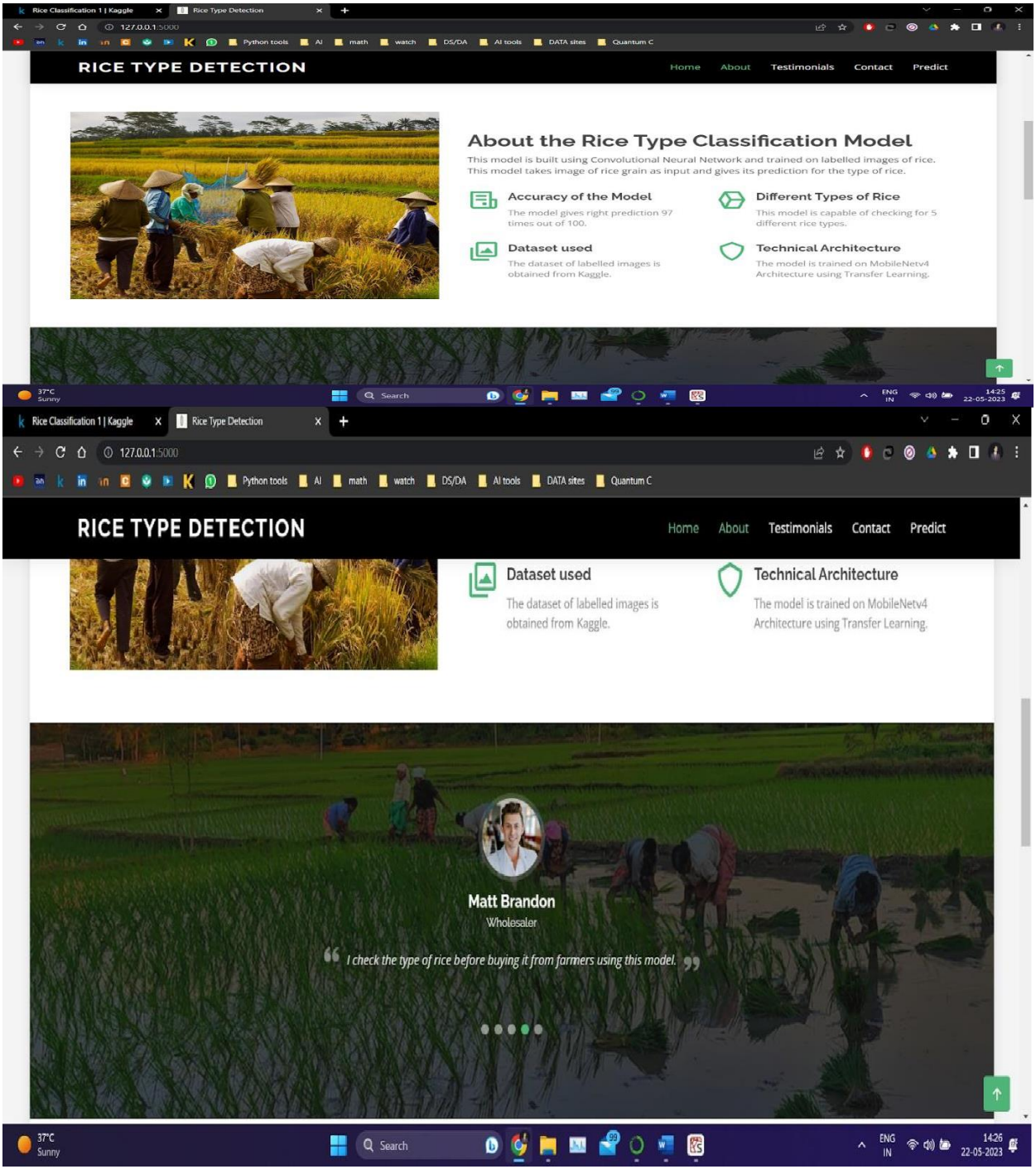
- All the sections below are included in the index.html page.
- Here the output is

:1.

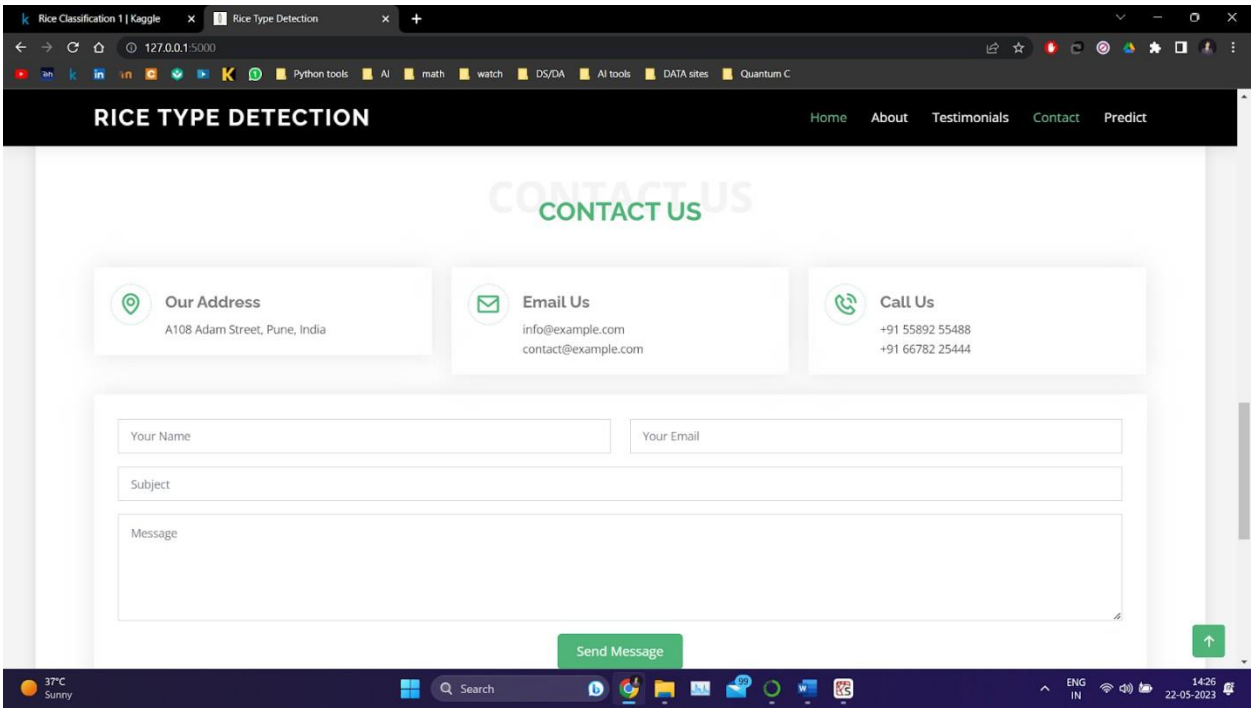




2.

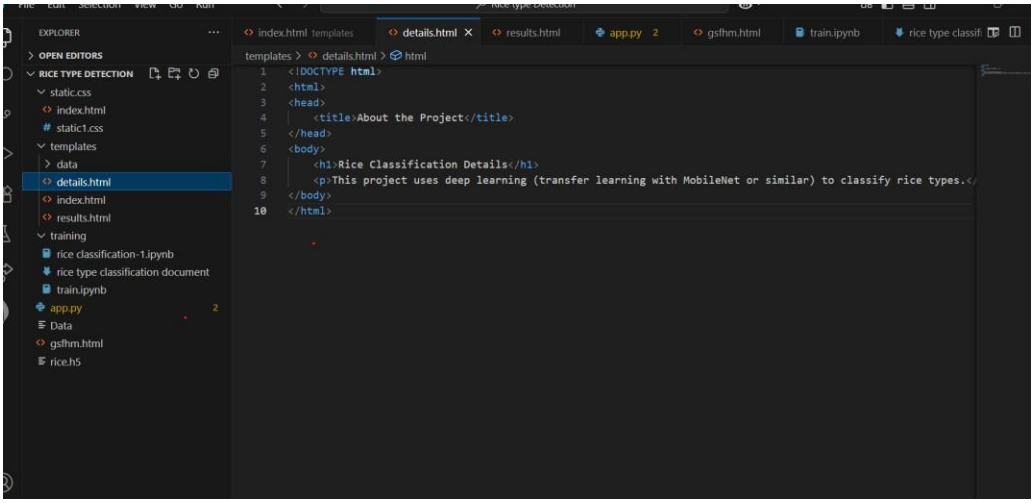

Rice Classification 1 | Kaggle
Rice Type Detection
127.0.0.1:5000
Python tools
AI
math
watch
DS/DA
AI tools
DATA sites
Quantum C
RICE TYPE DETECTION
Home
About
Testimonials
Contact
Predict
About the Rice Type Classification Model
This model is built using Convolutional Neural Network and trained on labelled images of rice.
This model takes image of rice grain as input and gives its prediction for the type of rice.
Accuracy of the Model
The model gives right prediction 97 times out of 100.
Different Types of Rice
This model is capable of checking for 5 different rice types.
Dataset used
The dataset of labelled images is obtained from Kaggle.
Technical Architecture
The model is trained on MobileNetv4 Architecture using Transfer Learning.
Matt Brandon
Wholesaler
I check the type of rice before buying it from farmers using this model.
37°C
Sunny
Search
ENG
IN
14:26
22-05-2023

- 3.



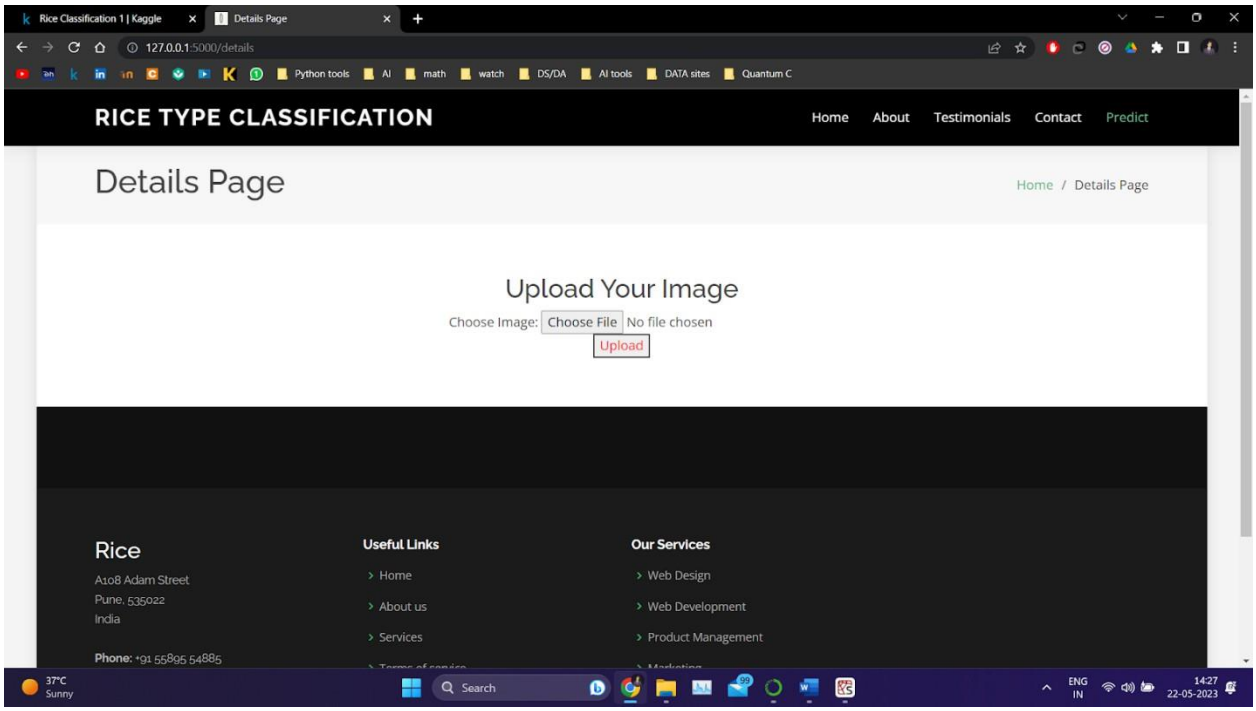
-

- Details. html

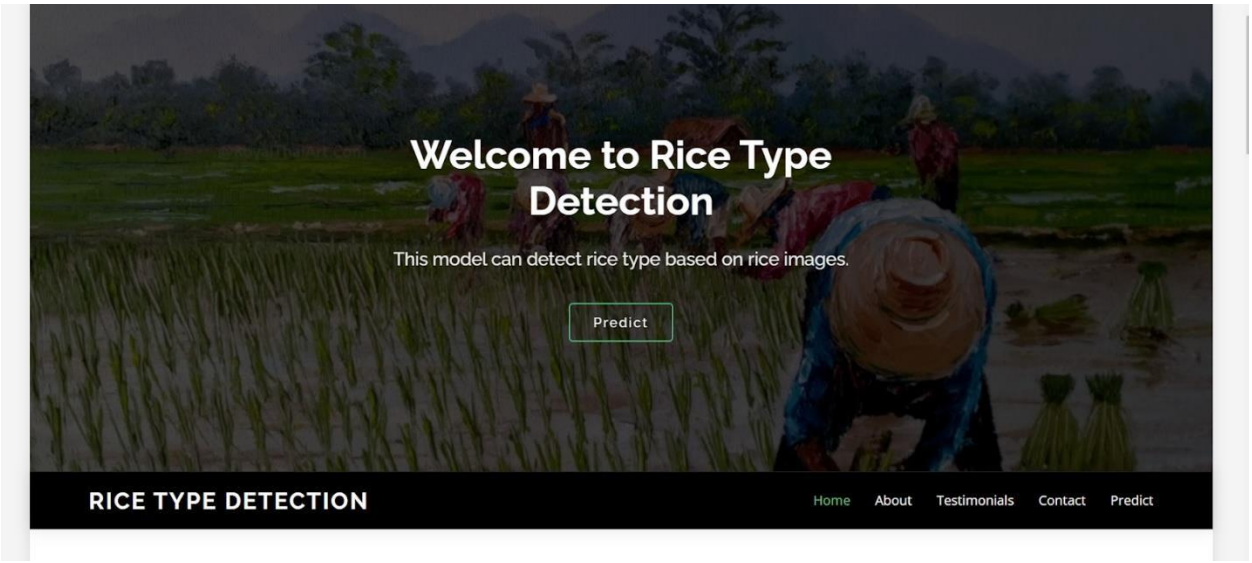


- When you click on the predict button, it will display the below page. You can test the model by passing a image

-

- 

- Results.html:



❖ **Known issues:**

❖ Here are the **known issues** commonly encountered in the **GrainPalette – Rice Type Classification Through Transfer Learning** project:

❖ ---

❖ **Known Issues in GrainPalette**

- **1. Model Prediction Delay**
- **Description:** The first prediction request can take time due to model loading.
- **Cause:** rice.h5 (MobileNetV2-based model) is loaded on-demand or during the first call.
- **Fix:** Load the model once at server start time and keep it in memory.

- **2. No Input Validation on Uploaded Files**
- **Description:** The /predict endpoint may crash or return generic errors if non-image files are uploaded.
- **Cause:** Missing MIME type validation or image format check.
- **Fix:** Add validation to check for valid image extensions (.jpg, .png, etc.) and MIME type.

- **3. Lack of Authentication**
- **Description:** Currently, no token/session authentication is enforced.
- **Risk:** Anyone can access /predict, leading to misuse or overload.
- **Fix:** Implement JWT or session-based authentication and protect sensitive endpoints.

- **4. Hardcoded Model Labels**
- **Description:** The class labels (e.g., Basmati, Jasmine) may be hardcoded or dependent on order during model training.
- **Issue:** If model is retrained or label mapping changes, predictions can become incorrect.
- **Fix:** Load label mapping dynamically using LabelEncoder pickle or consistent label file.

- **5. Limited Error Handling**

- **Description:** Backend often returns generic 500 errors or vague messages on failure.
- **Examples:**
- Missing file
- Corrupt image
- Prediction failure
- **Fix:** Improve error handling and return user-friendly, detailed error messages with 400, 422, 500 status codes.

- **6. CORS Restrictions (for Web Frontend)**
- **Description:** If accessed via a React frontend or browser, CORS errors may block the request.
- **Fix:** Use Flask-CORS and configure CORS(app, resources={r"/*": {"origins": "*"}}) (or restrict to frontend origin in production).

- **7. No GPU Support in Deployment**
- **Description:** The backend typically runs on CPU, which may slow predictions.
- **Fix:** Use GPU-accelerated environments (e.g., TensorFlow GPU with CUDA) for production deployment if scalability is needed.

- **8. No Logging for Monitoring or Debugging**
- **Description:** Prediction failures or API usage are not logged.
- **Fix:** Use Python's logging module or tools like Flask-Logging, Loguru, or integrate with logging services (ELK stack, Datadog, etc.).

- **9. Static Model — No Retraining/Improvement Loop**
- **Description:** Once deployed, the model cannot be updated with new data unless manually retrained offline.

- ❖ **Fix:** Consider adding an admin tool or retraining pipeline with new labeled images.
- ❖ ---
- ❖ **10. MobileNetV2 Limitations**
- ❖ **Description:** While MobileNetV2 is fast and efficient, it may struggle with subtle grain differences.
- ❖ **Fix:** Experiment with higher-capacity models like EfficientNet or ResNet if accuracy is a concern.
- ❖ ---
- ❖ Would you like this compiled into a section of the **project documentation PDF** under "Limitations and Known Issues"? I can also include a suggested **roadmap to resolve them** if needed.

➢ Future Enhancement :

💡 Future Enhancements

GrainPalette – A Deep Learning Odyssey In Rice Type Classification Through Transfer Learning

These enhancements aim to make the system more scalable, secure, accurate, and user-friendly in both research and production environments.

🚀 1. User Authentication & Role Management

- **Description:** Add JWT or session-based login for users.
- **Benefits:** Enables personalized usage, secure API access, and admin dashboards.
- **Enhancement Ideas:**
  - Admin/user roles
  - Rate limiting for anonymous users

- Secure login & logout flow

📄 2. Dataset Expansion & Model Improvement

- **Description:** Collect a larger and more diverse dataset of rice grains.
- **Benefits:** Improves model generalization and real-world accuracy.
- **Enhancement Ideas:**
  - Include more rice varieties
  - Use advanced models like **EfficientNet**, **ResNet50**, or **Swin Transformer**
  - Train using **AutoAugment** or **mixup** for better robustness

---

📱 3. Mobile App Integration

- **Description:** Build a companion mobile app using React Native or Flutter.
- **Benefits:** Allows users (e.g., farmers, traders) to classify rice directly from their phone camera.

🌐 4. Deploy to the Cloud

- **Description:** Host the model and API on platforms like **AWS**, **GCP**, **Azure**, or **Render**.
- **Benefits:** Global access, scalability, uptime monitoring.
- **Enhancement Ideas:**
  - Use GPU-enabled cloud instances
  - Add autoscaling based on demand

📊 5. Admin Dashboard & Analytics

- **Description:** Create a dashboard for monitoring predictions and system usage.
- **Benefits:** Helps in managing users, tracking uploads, and visualizing common rice types.
- **Features to Add:**
    - Total predictions over time
    - Most frequent rice types
    - User access logs

□ 6. Model Retraining from User Data

- **Description:** Allow users to submit incorrect predictions for review.
- **Benefits:** Can be used to fine-tune the model in the future using real-world edge cases.
- **Workflow:**
    - User reports incorrect prediction
    - Admin reviews and labels
    - Retrain periodically with confirmed new data

🌐 7. Multilingual Support

- **Description:** Add support for regional languages (e.g., Hindi, Telugu, Tamil) in the UI and messages.
- **Benefits:** Improves accessibility for farmers and local traders.

🎛️ 8. Image Preprocessing Optimization

- **Description:** Add real-time preprocessing for uploaded images (e.g., background removal, cropping).
- **Benefit:** Helps in improving model focus and performance, especially in noisy environments.

☐ 9. Confidence Score with Prediction

- **Description:** Return prediction probabilities along with the label.
- **Example Output:**

```
{
  "prediction": "Basmati",
  "confidence": 0.94
}
```

🗁 10. Batch Image Classification

- **Description:** Allow users to upload multiple images at once.
- **Use Case:** Traders or lab analysts can classify an entire batch efficiently.

☐ 11. Downloadable Reports

- **Description:** Let users download a PDF report of their classification result.
- **Includes:**
  - Prediction
  - Image preview
  - Timestamp
  - Confidence score

□ 12. Explainable AI (XAI)

- **Description:** Use Grad-CAM or similar techniques to highlight which part of the rice image influenced the prediction.

- **Benefit:** Makes model behavior more transparent and trustworthy.

🔗 13. API Versioning & Documentation Portal

- **Description:** Maintain API versioning (/v1, /v2) and serve Swagger/OpenAPI docs.
- **Benefit:** Developer-friendly API evolution with clear reference.

❖ ------THE END------