Project Report - Indian Food Classifier App

1. Executive Summary

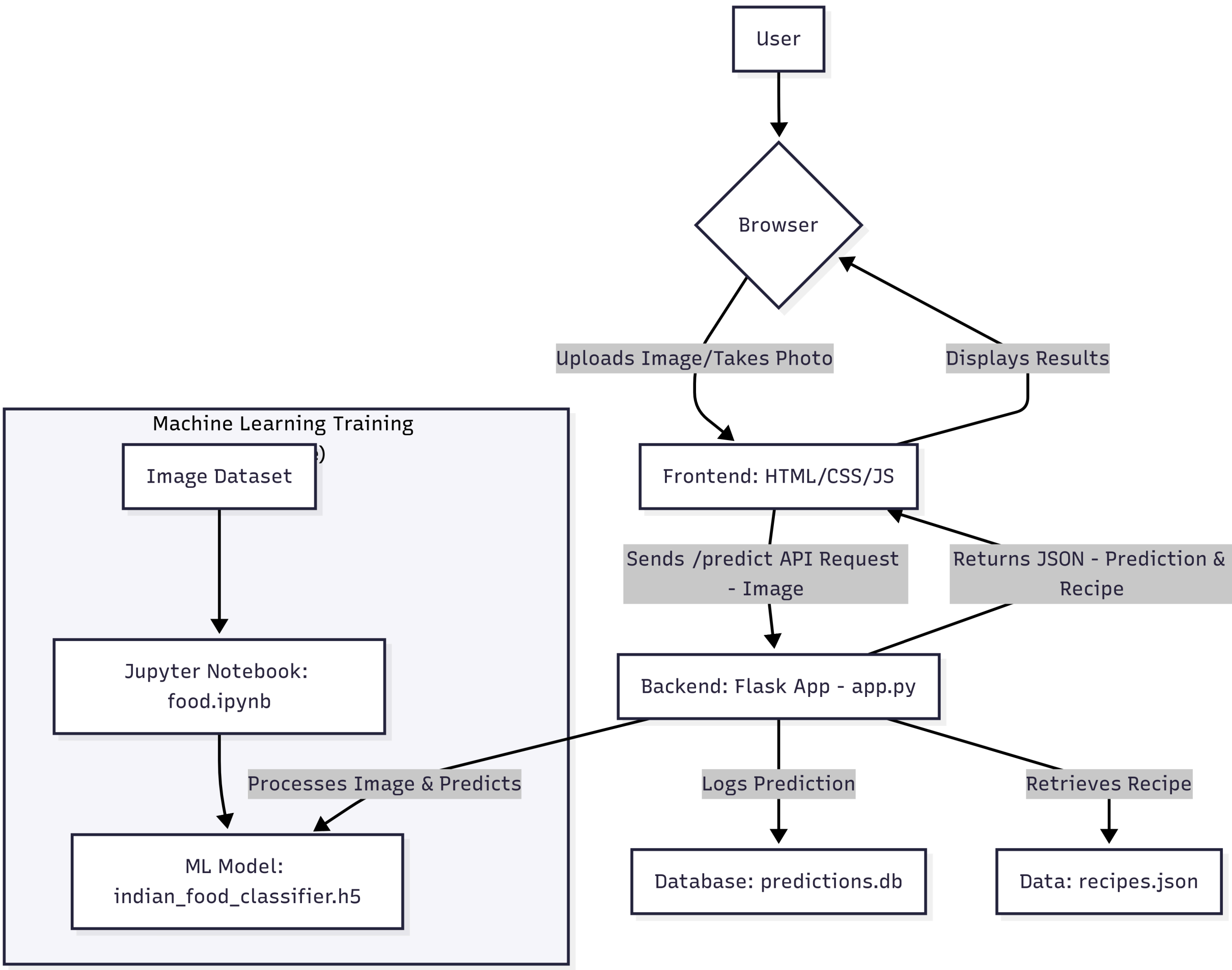
This report provides a detailed analysis of the Indian Food Classifier, a full-stack web application designed to identify Indian food dishes from an image and provide the corresponding recipe. The project consists of two main parts: a Machine Learning backend and a web-based frontend.

1. **Machine Learning Model (`food.ipynb`):** A Convolutional Neural Network (CNN) is trained using TensorFlow and Keras. It leverages **transfer learning** with the **MobileNetV2** architecture to classify images into one of 89 different Indian food categories. The notebook covers data loading, preprocessing, data augmentation, model training, and evaluation.
2. **Web Application (`app.py`, `index.html`, etc.):** A Flask-based web server provides the backend API. It receives an image from the user, processes it, feeds it to the trained model for prediction, logs the prediction in a SQLite database, retrieves the associated recipe from a JSON file, and returns the results to a dynamic, user-friendly frontend.

The final application allows users to either upload an image or use their device's camera to get an instant food identification and a detailed recipe, demonstrating a complete end-to-end Machine Learning project lifecycle.

2. Project Architecture

The application follows a classic client-server architecture:



- **Client (Browser):** The user interacts with the application through a web browser. The frontend, built with HTML, CSS, and JavaScript, handles all user-facing logic.
- **Server (Flask):** The Python-based Flask application acts as the brain. It serves the frontend files, exposes an API endpoint (`/predict`), and orchestrates the interaction between the ML model, database, and recipe data.
- **Model:** The pre-trained Keras model (`.h5` file) is loaded by Flask to perform the core prediction task.

3. Component Breakdown

3.1. Machine Learning Model (`food.ipynb`)

This Jupyter Notebook is responsible for creating the image classification model.

a. Data Loading and Preprocessing:

- The `tf.keras.preprocessing.image_dataset_from_directory` function is used to load the image dataset. This is highly efficient as it automatically infers class labels from the subdirectory names.
- The dataset is split into training (80%) and validation (20%) sets.

- Images are resized to a uniform `(224, 224)` pixels, which is the standard input size for the MobileNetV2 model.
- The data is loaded in batches of 32 for efficient memory usage during training.

b. Data Augmentation:

- A `tf.keras.Sequential` model is created with several data augmentation layers (`RandomFlip`, `RandomRotation`, `RandomZoom`, etc.).
- **Purpose:** This step is crucial for preventing **overfitting** and improving the model's ability to generalize. It creates modified versions of the training images on-the-fly, exposing the model to a wider variety of visual data (e.g., rotated, zoomed, or flipped images), making it more robust.

c. Model Architecture (Transfer Learning):

- **Base Model:** The project uses **MobileNetV2**, a state-of-the-art model pre-trained on the massive ImageNet dataset. MobileNetV2 is chosen for its high accuracy and computational efficiency, making it ideal for web applications.
- **Transfer Learning:** Instead of training a model from scratch, which would require vast amounts of data and time, the project uses the pre-trained MobileNetV2 as a feature extractor. Its `include_top=False` argument removes the original classification layers, and `base_model.trainable = False` freezes the weights of the pre-trained layers.
- **Custom Classifier Head:** On top of the frozen base model, a new "head" is added:
    1. `GlobalAveragePooling2D`: This layer reduces the spatial dimensions of the feature maps to a single vector per image, significantly reducing the number of parameters.
    2. `Dropout(0.3)`: This layer randomly sets 30% of its input units to 0 during training, another technique to prevent overfitting.
    3. `Dense(num_classes, activation='softmax')`: The final output layer. It has one neuron for each of the 89 food classes and uses the `softmax` activation function to output a probability distribution over the classes.

d. Training and Evaluation:

- The model is compiled with the `Adam` optimizer, `sparse_categorical_crossentropy` loss function (suitable for integer-based multi-class labels), and `accuracy` as the evaluation metric.
- The `model.fit()` function trains the model for 20 epochs.
- The training output shows that the training accuracy reaches ~71.5%, while the validation accuracy plateaus around ~52.5%. This gap indicates some overfitting, but data augmentation has helped mitigate a more severe divergence. The validation accuracy is still reasonably good for a complex 89-class problem.
- The history of training is plotted, visually confirming the accuracy and loss trends for both training and validation sets.

e. Saving Artifacts:

- The trained model is saved as `indian_food_classifier.h5`.
- The class names are saved to `class_names.txt` in the correct order, which is essential for the backend to correctly interpret the model's output.

3.2. Backend Application (`app.py`)

This is the Flask server that powers the application.

- **Initialization:**
  - It imports necessary libraries like `tensorflow`, `flask`, `PIL` (for image manipulation), `sqlite3`, and `json`.
  - On startup, it loads the trained model (`.h5` file), the class names, and the recipes from their respective files into memory. This is done once to avoid slow, repetitive loading on each request.
  - It initializes a SQLite database using `init_db()`, creating a `predictions` table if it doesn't exist. This table stores the path of the uploaded image, the predicted class, and a timestamp for every successful prediction.
- **Routing:**
  - `@app.route('/')`: This route serves the main `index.html` page to the user.
  - `@app.route('/predict', methods=['POST'])`: This is the core API endpoint. It only accepts POST requests, which are used for sending data (the image file) to the server.
- **Prediction Logic (`handle_prediction`)**
  1. **File Handling:** It checks if a file is present in the request. If not, it returns a 400 Bad Request error.
  2. **Image Saving & Processing:** The uploaded image is saved to the `static/uploads` folder with a unique timestamped filename. This allows the image to be potentially displayed back to the user or reviewed later.
  3. **Prediction:** The `predict_image` function takes the image, resizes it to `(224, 224)`, converts it to a NumPy array, and feeds it to the loaded `model`. It returns the class name with the highest predicted probability.
  4. **Database Logging:** It connects to the SQLite database and inserts a new record with the image path, predicted class, and current timestamp.
  5. **Recipe Retrieval:** It uses the predicted class name (e.g., `aloo_gobi`) as a key to look up the corresponding recipe from the `recipes_dict` loaded from `recipes.json`.
  6. **Response:** It constructs a JSON response containing the human-readable prediction name (e.g., "Aloo Gobi") and the detailed recipe information. This JSON is sent back to the frontend with a 200 OK status code.

- **Error Handling:** The function is wrapped in a `try...except` block to gracefully handle errors like invalid image files (`UnidentifiedImageError`) or other unexpected exceptions, returning informative JSON error messages to the frontend.

3.3. Frontend (HTML, CSS, JavaScript)

a. `index.html` (Structure):

- This file defines the entire user interface structure.
- **Header:** Contains the title and a brief description.
- **Mode Switcher:** Buttons to toggle between "Upload" and "Camera" modes.
- **Views:**
  - `#upload-view`: Contains the "drop zone" for file uploads.
  - `#camera-view`: Contains the `<video>` element for the camera feed and a "Snap Picture" button.
- **Preview:** `#image-preview-container` shows the uploaded or snapped image and the "Find Recipe" button.
- **Results:** `#results-section` is a container for the prediction and the detailed recipe, which is populated by JavaScript.
- **Spinner:** A loading spinner to provide feedback to the user while the backend is processing the request.

b. `style.css` (Styling):

- This file provides a modern, clean, and responsive design.
- **CSS Variables (`:root`):** Defines a color palette for easy theming and consistency.
- **Flexbox:** Used extensively for layout, ensuring elements are aligned and responsive.
- **Visual Feedback:** Provides `:hover` and `:active` states for buttons and the drop zone to improve user experience.
- **Animations:** A simple `@keyframes spin` animation is used for the loading spinner.
- **Responsive Design (`@media`):** Includes a media query to adjust padding and font sizes on smaller screens (mobile devices).

c. `static.js` (Logic):

- This is the most critical frontend file, handling all user interactions and API communication.
- **Event Listeners:** It sets up event listeners for all interactive elements: the mode-switcher buttons, the file input, the snap button, and the predict button.
- **Mode Switching:** The logic for hiding/showing the upload and camera views and managing the active state of the switcher buttons.

- **Camera API:**
  - `startCamera()`: Uses `navigator.mediaDevices.getUserMedia` to request access to the device's camera and streams the feed to the `<video>` element. It prioritizes the rear camera (`facingMode: 'environment'`).
  - `stopCamera()`: Halts all camera tracks to release the resource when not in use.
- **Image Handling:**
  - For uploads, it uses a `FileReader` to display a preview of the selected image.
  - For camera snaps, it draws the current video frame onto a hidden `<canvas>`, converts the canvas content to a `Blob` (a file-like object), and then sends this blob for prediction.
- **API Communication (`sendPredictionRequest`)**
  - This `async` function is the core of the client-server interaction.
  - It creates a `FormData` object to package the image file (or blob) for the POST request.
  - It uses the `fetch` API to send the request to the `/predict` endpoint.
  - It handles the response by parsing the JSON and dynamically updating the `resultDiv` and `recipeContainer` with the returned data.
  - It includes robust error handling to display meaningful error messages to the user if the API call fails.
  - It manages the visibility of the spinner and results sections to provide a smooth user experience.

4. How to Run the Project

1. **Prerequisites:**
   - Python 3.8+ and `pip`.
   - A local clone of the project repository.
2. **Setup:**
   - Navigate to the project directory in your terminal.
   - Create and activate a virtual environment (recommended):

     ```
     python -m venv venv
     # On Windows:
     venv\Scripts\activate
     # On macOS/Linux:
     source venv/bin/activate
     ```

   - Create a `requirements.txt` file with the following content:

     ```
     tensorflow
     Flask
     ```

```
Pillow
numpy
```

- Install the dependencies:

```
pip install -r requirements.txt
```

3. **File Structure:** Ensure all files are in the correct location:

```
.
├── app.py
├── food.ipynb
├── indian_food_classifier.h5
├── class_names.txt
├── recipes.json
├── predictions.db  (will be created on first run)
├── templates/
│   └── index.html
└── static/
    ├── style.css
    ├── static.js
    └── uploads/      (will be created on first run)
```

4. **Execution:**
   - Run the Flask application from the terminal:

```
python app.py
```

   - The server will start, typically on `http://127.0.0.1:5000` or `http://0.0.0.0:5000`.
   - Open this URL in your web browser to use the application.

5. Conclusion

The Indian Food Classifier is a well-structured, end-to-end project that successfully integrates a deep learning model into a functional web application. It demonstrates best practices like transfer learning, data augmentation, a clean client-server architecture, and a user-centric frontend design.