Flask App

1. **Flask**:

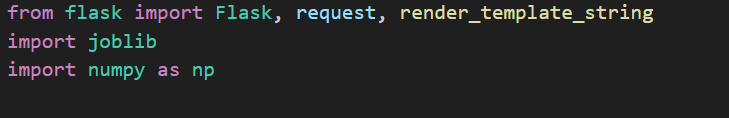
* **Purpose**: Flask is a lightweight web framework for Python. It allows you to build web applications quickly and with minimal setup.
* **Functions Used**:
  + Flask: Used to initialize the web application.
  + request: Allows you to access data submitted via an HTTP request (e.g., form data).
  + render\_template\_string: Allows you to render HTML templates within the Python code. Here, it’s used to display the prediction form and results on the web page.

1. **joblib**:

* **Purpose**: Joblib is a library for efficiently saving and loading large Python objects. It’s commonly used for saving machine learning models after training.
* **Functions Used**:
  + joblib.load: Used here to load the machine learning model and scaler (preprocessing object) that were saved previously. Loading these ensures that the web application can use the pre-trained model and scaler directly for predictions.

1. **numpy**:

* **Purpose**: Numpy is a library for numerical operations in Python, especially useful for handling arrays and performing mathematical computations.
* **Functions Used**:
  + np.array: Used here to organize the input features into a structured format (array) before passing them to the model.
  + reshape: Adjusts the shape of the array to match the model’s expected input format (a 2D array, i.e., one row of features).

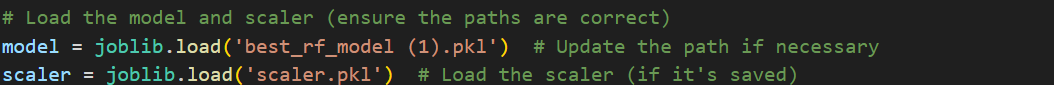


1. **Loading the Model**:

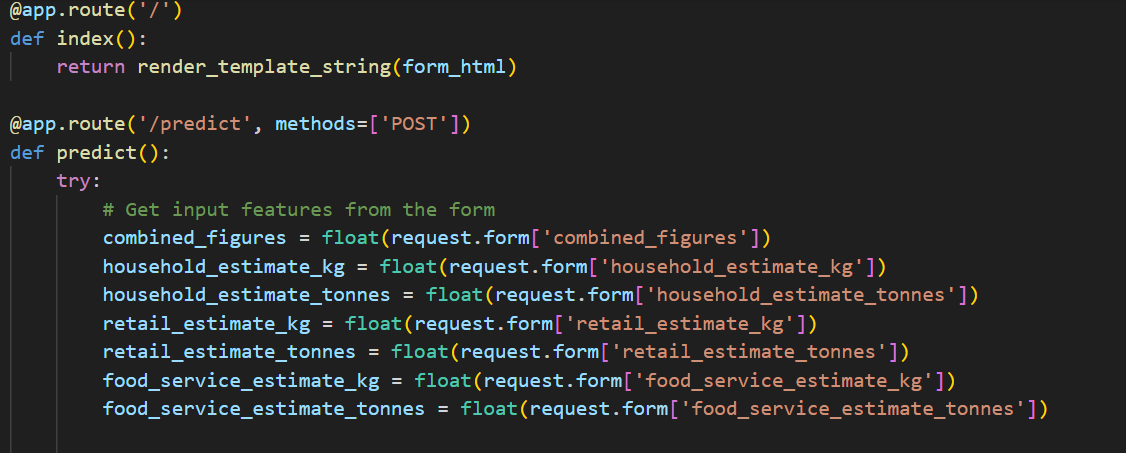
* model = joblib.load('best\_rf\_model (1).pkl')
* This line loads the machine learning model saved as 'best\_rf\_model (1).pkl' using joblib.load.
* The model could be a classifier or regressor trained in another script or notebook.
* Once loaded, model holds the trained model, allowing it to be used to make predictions directly within the Flask app.

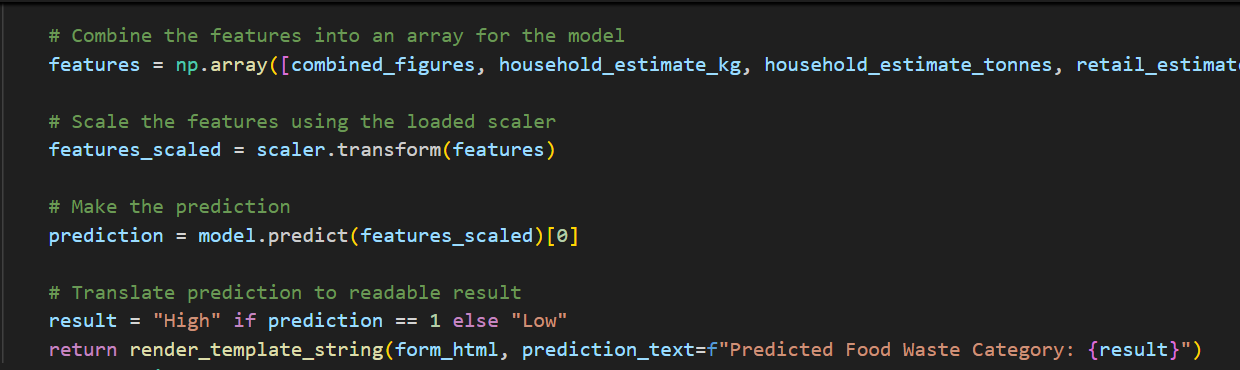
1. **Loading the Scaler**:

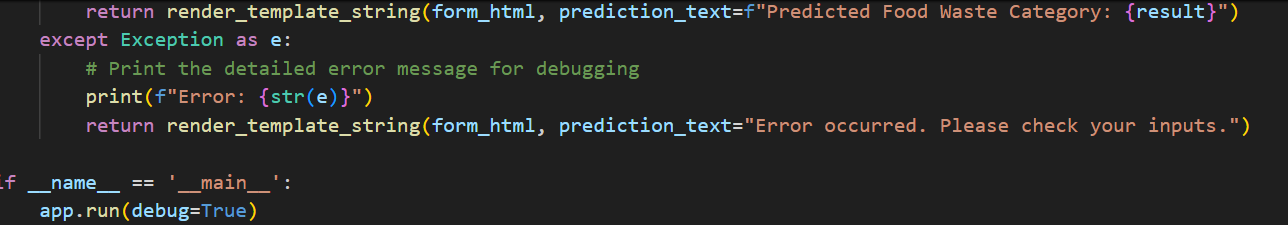
* scaler = joblib.load('scaler.pkl')
* Many machine learning models require input data to be scaled (e.g., using standardization or normalization) to achieve optimal performance. This is often done with a **scaler** during model training.
* In this code, the scaler object was saved to 'scaler.pkl', and it is now reloaded using joblib.load.
* The loaded scaler will transform new data in the same way as the training data, ensuring consistency in data preprocessing.



1. **Route Definitions**:
   * **@app.route('/') and def index()**:
     + This route (/) corresponds to the home page where the user will see the prediction form.
     + render\_template\_string(form\_html) dynamically renders the form's HTML (stored in form\_html) to the user.
   * **@app.route('/predict', methods=['POST']) and def predict()**:
     + The /predict route handles the form submission. It's defined to accept only POST requests, as it will receive data when the user submits the form.
     + request.form retrieves input values entered by the user in the form, each of which is converted to float for further processing.
2. **Form Data Extraction and Preprocessing**:
   * Input values for various features (combined\_figures, household\_estimate\_kg, etc.) are collected and converted to floats for compatibility with the model.
   * These values are then combined into a NumPy array (features), reshaped to ensure compatibility with the model input format.
3. **Data Scaling**:
   * The line features\_scaled = scaler.transform(features) uses the preloaded scaler (scaler) to scale the input data consistently with the data used for training. This step is essential to make sure the model sees data in a familiar range, avoiding skewed predictions.
4. **Model Prediction**:
   * The scaled features are passed to the model with prediction = model.predict(features\_scaled)[0].
   * The model’s output, prediction, is translated to a human-readable format: "High" for a value of 1 and "Low" for 0.
5. **Result Rendering**:
   * The result of the prediction is displayed back on the form page using render\_template\_string(form\_html, prediction\_text=f"Predicted Food Waste Category: {result}"). This renders the form again with an additional message showing the prediction result.
6. **Error Handling**:
   * If an error occurs (e.g., due to incorrect data types), the code will print the error message for debugging and display a user-friendly message on the page ("Error occurred. Please check your inputs.").
7. **Starting the Flask Application**:
   * if \_\_name\_\_ == '\_\_main\_\_': ensures the Flask app runs only when this script is executed directly.
   * app.run(debug=True) launches the application in debug mode, which provides useful error messages and automatically reloads the server when changes are made.

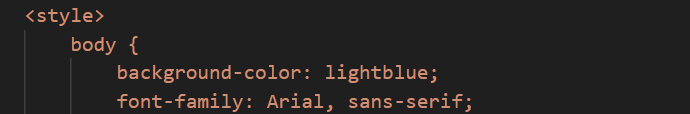






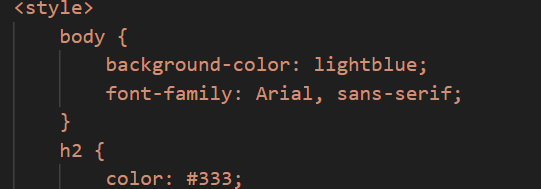
1. **Background Color**:

* background-color: lightblue; in the body selector sets the entire page’s background color to a soft, light blue, creating a calm, clean look for the form.



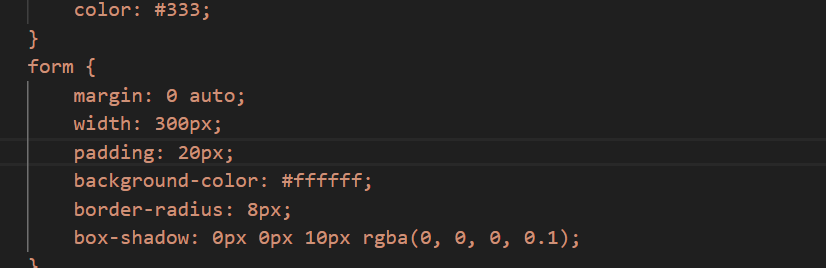
1. **Font and Heading Styling**:

* font-family: Arial, sans-serif; in the body selector sets the text style across the entire page, making the font clean and easy to read.
* h2 styling (color: #333;) changes the color of headings to a dark gray, which contrasts well with the light blue background without being harsh.



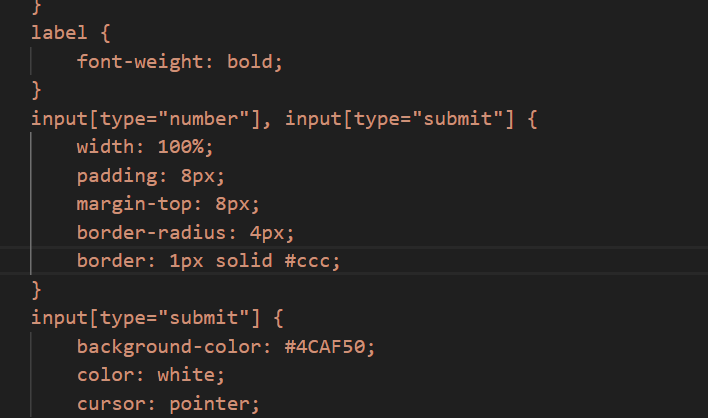
1. **Form Container**:

* margin: 0 auto; centers the form horizontally.
* width: 300px; defines the form’s width, keeping it compact and focused.
* padding: 20px; adds spacing inside the form for readability.
* background-color: #ffffff; sets the form’s background to white, standing out against the light blue page.
* border-radius: 8px; rounds the form’s corners, creating a soft, modern look.
* box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1); applies a light shadow to add depth, making the form visually distinct from the background.



1. **Labels and Input Fields**:

* font-weight: bold; in the label selector emphasizes the field names, making them more noticeable.
* width: 100%; for input fields ensures they take up the full width of the form, creating alignment.
* padding: 8px; adds spacing within input fields for comfort.
* border-radius: 4px; and border: 1px solid #ccc; make input fields visually consistent and easy to use.



1. **Submit Button**:

* background-color: #4CAF50; color: white; styles the button with a green background and white text.
* cursor: pointer; changes the mouse pointer to indicate a clickable button.
* input[type="submit"]:hover { background-color: #45a049; } adjusts the button color on hover, making it slightly darker for interactive feedback

