FLASK APP:

```
from flask import Flask, render_template, request
import pandas as pd
import joblib
import ast
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
```

- This block imports the necessary libraries and modules for running the Flask web application, data manipulation with pandas, loading the trained model using joblib, parsing strings as Python literals using ast, natural language processing functionalities from nltk, and the TF-IDF vectorizer from scikit-learn.

```
nltk.download('stopwords')
```

- This line downloads the stopwords corpus from nltk if it is not already present in the system. Stopwords are commonly used words (e.g., “the,” “is,” “and”) that are often removed from text data during natural language processing tasks.

```
app = Flask(__name__)
```

- This line creates a Flask application instance.

```
model = joblib.load('clustering_model.pkl')
print('\n Model loaded \n')
```

- This code block loads the trained clustering model from the ‘clustering_model.pkl’ file using joblib’s `load()` function. The model will be used for recommending recipes based on user input.

```
df = pd.read_csv("cleaned.csv")  # Replace with the path to your cleaned dataset
print('\n Dataset read \n')
```

- This code block reads the cleaned dataset from a CSV file named “cleaned.csv” using pandas’ `read_csv()` function. The dataset contains recipe information and will be used for recommending recipes.

```
vectorizer = TfidfVectorizer()
```

- This line initializes a TF-IDF vectorizer from scikit-learn's `TfidfVectorizer` class. The vectorizer will be used to convert the recipe descriptions into numerical feature vectors based on the Term Frequency-Inverse Document Frequency (TF-IDF) algorithm.

```
df['Features'] = df['Features'].apply(ast.literal_eval)
df['ingredients'] = df['ingredients'].apply(ast.literal_eval)
df['directions'] = df['directions'].apply(ast.literal_eval)
print('\n Literal eval done \n')
```

- This code block applies the `ast.literal_eval()` function to each value in the 'Features,' 'ingredients,' and 'directions' columns of the DataFrame. It converts the string representations of lists into actual list objects. This step is necessary to process the data properly for further analysis.

```
all_features = [feature for features in df['Features'] for feature in features]
```

- This line creates a list `all_features` by iterating over each list of features in the 'Features' column of the DataFrame and extracting individual features. This creates a consolidated list of all the features mentioned in the recipes.

```
nltk_stopwords = set(stopwords.words('english'))
more_stops = ['hello', 'hi', 'hey', 'good', 'morning', 'evening']  # Add your desired stop words he
re
stopwords = nltk_stopwords.union(more_stops)
all_features = [feature for feature in all_features if feature.lower() not in stopwords]
print('\n stopwords removed from all_ingredients \n')
```

- This code block creates a set of stopwords using the stopwords corpus from nltk. Additional custom stopwords are added to the set.
- The `all_features` list is then filtered to remove any features that are present in the stopwords set. This step removes common words that are unlikely to contribute to the clustering and recommendation process.

```
documents = df['RecipeCategory'].tolist()
```

- This line extracts the 'RecipeCategory' column
  from the DataFrame and converts it to a list named `documents`. This list will be used as input for creating the TF-IDF matrix.

```
tfidf_matrix = vectorizer.fit_transform(documents)
print('\n tfidf-matrix created \n')
```

- This code block applies the TF-IDF vectorizer to the `documents` list, transforming the text data into a sparse matrix representation where each row represents a document and each column represents a feature. The resulting matrix is assigned to the variable `tfidf_matrix`. This matrix will be used for clustering and recommending recipes.

```
def recommend_recipes(user_input):
    # Perform clustering on user input
    user_tfidf = vectorizer.transform([user_input])
    label = model.predict(user_tfidf)

    # Filter recipes based on the predicted cluster label
    cluster_labels = model.predict(tfidf_matrix)
    cluster_recipes = df[cluster_labels == label[0]]

    # Check if there are any recipes in the cluster
    if cluster_recipes.empty:
        return None

    # Calculate confidence scores for each recipe based on keyword matching
    recipe_scores = {}
    user_keywords = set(user_input.lower().split(','))

    for index, recipe in cluster_recipes.iterrows():
        features = set(recipe['Features'])
        matching_keywords = user_keywords.intersection(features)
        confidence_score = len(matching_keywords) / len(user_keywords)
        recipe_scores[index] = confidence_score

    # Sort recipes based on confidence scores in descending order
    sorted_recipes = sorted(recipe_scores, key=recipe_scores.get, reverse=True)

    # Select top 5 recipes with the highest confidence scores
    recommended_recipes = cluster_recipes.loc[sorted_recipes[:5]]

    return recommended_recipes
```

- This code block defines a function `recommend_recipes(user_input)` that takes user input as an argument and recommends recipes based on that input.
- The function first performs clustering on the user input by transforming it into a TF-IDF vector using the `vectorizer` and predicting its cluster label using the `model`.
- It then filters the recipes in the DataFrame `df` based on the predicted cluster label, storing the filtered recipes in `cluster_recipes`.
- Next, the function calculates a confidence score for each recipe in `cluster_recipes` by counting the number of matching keywords between the user input and the recipe features. The scores are stored in the `recipe_scores` dictionary.
- The recipes in `cluster_recipes` are sorted based on the confidence scores in descending order, and the top 5 recipes with the highest scores are selected and returned as `recommended_recipes`.

```
chat_history = []
```

- This line initializes an empty list named `chat_history` that will store the conversation history between the user and the chatbot.

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

- This code block creates a route decorator for the default route `'/'`. When a user accesses the website's root URL, the function `index()` is triggered. It returns the rendered template file `'index.html'`, which will be displayed in the user's browser.

```
@app.route('/recommend', methods=['POST'])
def recommend():
    user_input = request.form['user_input']

    # Get recipe recommendations based on user input
    recommendations = recommend_recipes(user_input)

    # ...
```

- This code block creates a route decorator for the route `'/recommend'` with the HTTP method `'POST'`. When the user submits a form with `method="POST"` to the `'/recommend'` URL, the function `recommend()` is triggered.
- The user's input from the form is

obtained from `request.form['user_input']`, and the `recommend_recipes()` function is called to get recipe recommendations based on the user input.

```
# Check if the user input contains multiple words without a comma
    if len(user_input.split(',')) > 1 and ',' not in user_input:
        return render_template('recommendations.html',
                               chat_history=[{'USER': user_input,
                                              'BOT': 'Please separate multiple words with a comm
a.'}],
                               recommendations=pd.DataFrame().head())
```

- This code block checks if the user input contains multiple words without a comma. If so, it renders the template file `'recommendations.html'` and displays a message in the `chat_history` that instructs the user to separate multiple words with a comma. The recommendations section will be empty (`pd.DataFrame().head()`).

```
# Filter rows where any of the strings in 'Features' column contain the user input
    matching_keywords = []
    for word in user_input.split(','):
        if any(word.strip().lower() in feature.lower() for feature in all_features):
            matching_keywords.append(word)

    if not matching_keywords or any(word.strip().lower() in stopwords for word in user_input.split
(',')):
        return render_template('recommendations.html',
                               chat_history=[{'USER': user_input,
                                              'BOT': 'No recipe found. Try again.'}],
                               recommendations=pd.DataFrame().head())
```

- This code block filters the rows of the DataFrame based on whether any of the strings in the 'Features' column contain the user input. It checks each word in the user input (after splitting on commas) and searches for matches in the `all_features` list.
- If no matching keywords are found or if any of the words in the user input are in the stopwords set, it renders the `'recommendations.html'` template and displays a message in the `chat_history` indicating that no recipe was found. The recommendations section will be empty (`pd.DataFrame().head()`).

```
# Get the recipe names for chat history display
    recipe_names = recommendations['title'].tolist()
```

```
    # Add the user input and recipe names to the chat history
    conversation = {'USER': user_input, 'BOT': dict(enumerate(recipe_names, 1))}
    chat_history.append(conversation)  # Add recommended recipes to chat history

    # Store the chat history in cookies (limited to the last 5 items)
    response = app.make_response(
        render_template('recommendations.html', recommendations=recommendations, chat_history=chat_
history[-5:]))

    return response
```

- This code block gets the recipe names from the recommendations DataFrame and converts them to a list called `recipe_names`.
- It creates a dictionary `conversation` containing the user input and the recipe names enumerated from 1. The conversation is appended to the `chat_history`.
- The `chat_history` is stored in cookies, limited to the last 5 items.
- Finally, the template `'recommendations.html'` is rendered with the `recommendations`, `chat_history`, and `response` objects.

```
if __name__ == '__main__':
    app.run()
```

- This line checks if the script is being run directly (not imported as a module). If so, it starts the Flask application by calling the `run()` method on the `app` instance, and the web application becomes accessible on the local server.

Here's the significance of each code block and why it was used:

1. Importing Required Libraries:
    - `from flask import Flask, render_template, request` : Flask is a micro web framework used for building web applications. It provides tools for routing, rendering templates, and handling HTTP requests.
    - `import pandas as pd` : Pandas is a powerful data manipulation library used for working with structured data.
    - `import joblib` : Joblib is used for loading the clustering model from a file.
    - `import ast` : The `ast` module is used for safely evaluating literal expressions in strings.

- `import nltk` : NLTK (Natural Language Toolkit) is a library used for natural language processing tasks.
- `from nltk.corpus import stopwords` : NLTK's `stopwords` corpus contains a list of common words that can be removed from text as they do not carry significant meaning.
- `from sklearn.feature_extraction.text import TfidfVectorizer` : The `TfidfVectorizer` class from scikit-learn is used to convert text documents into a numerical matrix representation using the TF-IDF (Term Frequency-Inverse Document Frequency) technique.

2. Downloading Stopwords:

- `nltk.download('stopwords')` : This line downloads the stopwords corpus from NLTK if it hasn't been downloaded already. Stopwords will later be used for removing common words from the text data.

3. Flask Application Setup:

- `app = Flask(__name__)` : Creates an instance of the Flask application.

4. Loading the Clustering Model:

- `model = joblib.load('clustering_model.pkl')` : Loads the clustering model from the 'clustering_model.pkl' file. The model is trained to cluster recipes based on their features.

5. Loading the Dataset:

- `df = pd.read_csv("cleaned.csv")` : Reads the cleaned dataset file ('cleaned.csv') into a DataFrame. The dataset contains information about recipes.

6. Creating a TF-IDF Vectorizer:

- `vectorizer = TfidfVectorizer()` : Initializes a TF-IDF vectorizer, which will be used to convert text data into a numerical representation suitable for clustering.

7. Applying Literal Evaluation:

- `df['Features'] = df['Features'].apply(ast.literal_eval)` : Applies the `ast.literal_eval` function to the 'Features' column of the DataFrame, which converts the string representation of a list into an actual list object.
- `df['ingredients'] = df['ingredients'].apply(ast.literal_eval)` : Applies `ast.literal_eval` to the 'ingredients' column of the DataFrame.
- `df['directions'] = df['directions'].apply(ast.literal_eval)` : Applies `ast.literal_eval` to the 'directions' column of the DataFrame.
- The literal evaluation is performed to convert the string representations of lists into usable list objects.

8. Removing Stopwords:

- `all_features = [feature for features in df['Features'] for feature in features]` : Creates a list of all features extracted from the 'Features' column of the DataFrame.
- `nltk_stopwords = set(stopwords.words('english'))` : Retrieves the set of English stopwords from NLTK.
- `more_stops = ['hello', 'hi', 'hey', 'good', 'morning', 'evening']` : Additional words to be treated as stopwords.
- `stopwords = nltk_stopwords.union(more_stops)` : Combines the NLTK stopwords and additional stopwords into a single set.
- `all_features = [feature for feature in all_features if feature.lower() not in stopwords]` : Filters the `all_features` list by removing stopwords.
- This step removes common and less informative words from the `all_features` list.

9. Initializing the Documents List:

   - `documents = df['RecipeCategory'].tolist()` : Creates a list of recipe categories from the 'RecipeCategory' column of the DataFrame.
   - This list will be used for generating the TF-IDF matrix.

10. Creating the TF-IDF Matrix:

    - `tfidf_matrix = vectorizer.fit_transform(documents)` : Applies the TF-IDF vectorizer to the `documents` list, which converts the text data into a numerical matrix representation.
    - This matrix will be used for clustering and calculating recipe similarities.

11. Defining the `recommend_recipes()` Function:

    - This function takes user input, performs clustering on it, filters recipes based on the predicted cluster label, calculates confidence scores for each recipe, sorts the recipes based on confidence scores, and selects the top 5 recommended recipes.

12. Flask Routes:

    - `@app.route('/')` : Decorator for the root URL, which renders the 'index.html' template.
    - `@app.route('/recommend', methods=['POST'])` : Decorator for the '/recommend' URL with POST method, which handles the user input from the form submission and calls the `recommend_recipes()` function to get recipe recommendations.

13. Handling User Input:

    - The `/recommend` route receives the user input from the form using `request.form['user_input']` .
    - It checks if the user input contains multiple words without a comma, and if so, renders the template with an appropriate message.

- It filters the rows based on matching keywords in the 'Features' column and stopwords, and renders the template with an appropriate message if no recipe is found.

14. Generating Chat History and Recommendations:

    - The recipe names from the recommendations DataFrame are obtained and added to the `chat_history`.
    - The `chat_history` is stored in cookies and limited to the last 5 items.
    - The recommendations and chat history are rendered in the 'recommendations.html' template.

15. Running the Application:

    - `if __name__ == '__main__':` : This condition checks if the script is being run directly.
    - `app.run()` : Starts the Flask application and makes it accessible on the local server.

Overall, this code sets up a Flask web application that takes user input, performs clustering, and provides recipe recommendations based on the user's input. It uses Pandas for data manipulation, NLTK for natural language processing tasks, and scikit-learn for vectorizing text data and loading the clustering model. The code implements various data preprocessing steps, filters, and sorts the recipes based on similarity, and provides an interactive interface for users to receive recipe recommendations.