Streamlit_lecnote

May 23, 2025

1 App Interface Design with Streamlit

[1]:

```
import streamlit as st              # Build web-based front-end interface
import requests                     # Send HTTP requests (e.g., API calls)
import pandas as pd                 # Handle tabular data (commonly with␣
 ↪DataFrame)
import numpy as np                  # Numerical computing with array and matrix␣
 ↪support
import matplotlib.pyplot as plt     # Plotting library for static visualizations
import json                         # Parse and store JSON data
import os                           # OS-level operations like file paths
import difflib                      # Compute string similarity (e.g., fuzzy␣
 ↪matching)

API_KEY = "nqj9Kh3QVKwI4AFfuwGddoSOQznWReylbYLFynzU" # This is the API key to␣
 ↪the USDA API
```

1.1 Part 1 Logo & Setting up / Initializing User Data Base

1.1.1 Code Explanation

The `USER_DB` initialization block handles loading and preparing user data for the application.

1. A variable `logo_url` stores the URL of an image icon used in the app's header, which will be displayed in the Streamlit interface.
2. A variable `USER_DB_PATH` is assigned the value `"user_db.json"` to specify the local file used for saving and loading user data.
3. The script checks if the file at `USER_DB_PATH` exists using `os.path.exists()`:
   - If it exists, the file is opened in read mode and parsed using `json.load(f)` to populate the `USER_DB` dictionary.
   - If it does not exist, a default dictionary `USER_DB` is created manually with sample users.
4. Each user in `USER_DB` (e.g., `"alice"` and `"bob"`) contains the following fields:
   - `"password"`: a plain text password for login
   - `"gender"`: either `"male"` or `"female"`
   - `"age"`: numeric value in years
   - `"height"`: height in centimeters

- "weight": weight in kilograms
- "activity_level": one of "inactive", "low active", "active", or "very active"
- "goal": either "fat_loss" or "muscle_gain" to indicate their objective

[2]:

```
# --- Logo Link ---
logo_url = "https://cdn-icons-png.flaticon.com/512/590/590685.png"  # Logo URL␣
↪used in Streamlit header

# --- File path for user DB persistence ---
USER_DB_PATH = "user_db.json"  # Local file to persist user data

# --- Load user DB from file if exists ---
if os.path.exists(USER_DB_PATH):
    with open(USER_DB_PATH, "r") as f:
        USER_DB = json.load(f)             # Load user data from JSON file
else:
    USER_DB = {                            # If file doesn't exist, initialize␣
↪in-memory user DB
        "alice": {
            "password": "1234",            # Simple password (not secure for␣
↪real apps)
            "gender": "female",
            "age": 28,
            "height": 160,                 # in cm
            "weight": 55,                  # in kg
            "activity_level": "active",    # User-reported activity level
            "goal": "fat_loss"             # Goal: either "fat_loss" or␣
↪"muscle_gain"
        },
        "bob": {
            "password": "5678",
            "gender": "male",
            "age": 30,
            "height": 175,
            "weight": 70,
            "activity_level": "inactive",
            "goal": "muscle_gain"
        }
    }
```

1.1.2 Further Explanation for Lines of Code that could be Confusing

Line 9

1. `open(USER_DB_PATH, "r")`

   `open()` is a built-in Python function used to open a file.

   `"r"` is the mode, which stands for read. It means you want to read the file, not write to or modify it.

USER_DB_PATH is the file path — in this case, "user_db.json", which is a JSON file used to store user data.

2. with ... as f:

   This is Python's with statement, which ensures the file is properly closed after being opened — even if an error occurs.

   f is the name given to the file object. You can use it to read the contents of the file.

1.2 Part 2 Laying Out the Page and Setting What Goes to the Header

1.2.1 Code Explanation

The Streamlit setup and header display logic ensures a polished layout with branding.

1. The st.set_page_config(layout="centered") function configures the Streamlit page layout so that all content is centered. This enhances visual balance and makes the app look more professional on all screen sizes.
2. The line col_logo, col_title = st.columns([2, 6]) creates a layout with two columns using Streamlit's layout API. The first column (1/4 width) is for the logo, and the second column (3/4 width) is for the title and subtitle text.
3. Inside the with col_logo: block, the logo image stored at logo_url is rendered using st.image(), scaled to a width of 150 pixels to fit cleanly into the layout.
4. The with col_title: block renders the following elements:
   - st.markdown("## Nutrition Scoring App 2.5.3"): A markdown-based header for the app title.
   - st.caption("Your personalized guide to smarter food choices!"): A short descriptive subtitle under the title.
   - st.caption("A Python Project Created by Group 02 with Python and Streamlit"): A final caption crediting the development team and tools used.

[ ]:

```
# --- Streamlit page setup ---
st.set_page_config(layout="centered")  # Set layout to centered (better visual␣
 ↪balance)

# --- Header with logo aligned to title ---
col_logo, col_title = st.columns([2, 6])  # Two columns: logo (1/4 width), title␣
 ↪(3/4 width)

with col_logo:
    st.image(logo_url, width=150)  # Display logo image at defined width

with col_title:
    st.markdown("## Nutrition Scoring App 2.5.3")  # Main title (Markdown style)
    st.caption("Your personalized guide to smarter food choices!")  # Subtitle␣
 ↪or tagline
```

```
    st.caption("A Python Project Created by Group 02 with Python and Streamlit") # Credit line
```

1.3 Part 3 Logging in

1.3.1 Code Explanation

The `login/register` interface uses Streamlit session state to manage user authentication and persist login data across interactions.

1. The code first checks whether the key `"logged_in"` exists in `st.session_state`. If it doesn't, it initializes the login state by setting `st.session_state.logged_in = False`.
2. If the user is not logged in, the app displays a login/registration form. A radio button `st.radio()` lets the user choose between `"Login"` and `"Register"` modes.
3. In `"Login"` mode:
   - Two text input fields collect the `username` and `password`, with password input masked.
   - When the `"Login"` button is clicked:
     - It verifies the entered credentials by checking if the `username` exists in `USER_DB` and the stored password matches.
     - If valid, it sets three session state variables: `logged_in`, `user_profile`, and `username`, and triggers a rerun using `st.rerun()`.
     - If the login fails, it shows an error message using `st.error()`.
4. In `"Register"` mode:
   - Three password fields are displayed to input a `username`, `new_pass`, and `confirm_pass`.
   - Additional profile information is collected using input fields and dropdowns:
     - `gender`, `age`, `height`, `weight`, `activity_level`, and `goal`.
5. Upon clicking `"Register"`:
   - The program performs a series of checks:
     - If the `username` already exists in `USER_DB`, show a username conflict error.
     - If `new_pass` and `confirm_pass` do not match, show a password mismatch error.
     - If the username is shorter than 3 characters or password shorter than 4, show a warning.
   - If all checks pass:
     - A new user record is added to `USER_DB` with all profile info.
     - It tries to save `USER_DB` to disk using `json.dump()` inside a `try-except` block.
     - On success, it updates the session state and displays a success message.
6. Finally, `st.stop()` is called to prevent the rest of the app from rendering unless the user has successfully logged in.

[ ]:

```
# --- Track login state ---
if "logged_in" not in st.session_state:
    st.session_state.logged_in = False  # Initialize login state

# --- Show login/register form if not logged in ---
```

```
if not st.session_state.logged_in:
    st.markdown("##  Member Access")
    auth_mode = st.radio("Choose action", ["Login", "Register"])

    # --- Login form ---
    if auth_mode == "Login": # What shows if user choose the Login action
        username = st.text_input("Username")
        password = st.text_input("Password", type="password")

        if st.button("Login"): # What happens next if user hits the Login button
            if username in USER_DB and USER_DB[username]["password"] == password:
                st.session_state.logged_in = True # Change the session state␣
 ↪from not logged in to logged in
                st.session_state.user_profile = USER_DB[username]  # Load full␣
 ↪user data
                st.session_state.username = username
                st.rerun()
            else:
                st.error(" Invalid username or password")

    # --- Registration form ---
    elif auth_mode == "Register": # What shows if user choose the Register action
        new_user = st.text_input("Choose a username(at least 3 characters)")
        new_pass = st.text_input("Create a password(at least 4 characters)",␣
 ↪type="password")
        confirm_pass = st.text_input("Confirm password(at least 4 characters)",␣
 ↪type="password")

        # Profile information fields
        st.markdown("###  Profile Info")
        col1, col2 = st.columns(2)
        with col1:
            gender = st.selectbox("Biological Sex", ["male", "female"])
            age = st.number_input("Age", 1, 99, 25)
            height = st.number_input("Height (cm)", 100, 250, 165)
        with col2:
            weight = st.number_input("Weight (kg)", 30, 150, 60)
            activity_level = st.selectbox("Activity level", ["inactive", "low␣
 ↪active", "active", "very active"])
            goal = st.selectbox("Goal", ["muscle_gain", "fat_loss"])

        # Validation and account creation
        if st.button("Register"):
            if new_user in USER_DB: # Username has to be unique
                st.error(" Username already taken.")
            elif new_pass != confirm_pass: # Password must match to confirm the␣
 ↪user correctly typed in his/her desired password
```

```
                st.error(" Passwords do not match.")
            elif len(new_user) < 3 or len(new_pass) < 4:
                st.warning(" Username must be 3+ characters, password 4+.")
            else:
                # Save new user data by updating the previous user database
                USER_DB[new_user] = {
                    "password": new_pass,
                    "gender": gender,
                    "age": age,
                    "height": height,
                    "weight": weight,
                    "activity_level": activity_level,
                    "goal": goal
                }

                try:
                    with open(USER_DB_PATH, "w") as f:
                        json.dump(USER_DB, f, indent=4)
                        print(" Saved USER_DB")
                except Exception as e:
                    st.error(f" Error saving user: {e}")
                    return

                st.session_state.logged_in = True # Change the session state from not logged in to logged in
                st.session_state.user_profile = USER_DB[new_user] # Log in with the new user's profile
                st.session_state.username = new_user
                st.success(" Registration successful! Logging you in...")
                st.rerun()

    st.stop()  # Prevent rendering other UI before login
```

1.3.2 Further Explanation for Lines of Code that could be Confusing

Line 62 to 68 The purpose of try and except is to prevent your app from crashing if saving fails.

1. `try:`
   Starts a try block — this is where we put code that *might fail.*

2. `with open(USER_DB_PATH, "w") as f:`
   Tries to open the file defined by `USER_DB_PATH` in **write mode** (`"w"`).

   - If the file doesn't exist, Python will create it.
   - If it exists, Python will **overwrite** it.

3. `json.dump(USER_DB, f, indent=4)`

   The `json.dump()` is a method from Python's built-in json module. It writes a Python object (like a dictionary or list) to a file in JSON format.

In this case, it takes the `USER_DB` Python dictionary and **saves it in JSON format** to the file `f`.

  - `indent=4` makes it human-readable (pretty printed).

4. `print(" Saved USER_DB")`
   Shows a message in the terminal (not Streamlit) that saving succeeded.

5. `except Exception as e:`
   If anything goes wrong (like the path doesn't exist), Python jumps here.

6. `st.error(f" Error saving user: {e}")`
   Shows an error message in the **Streamlit app UI**, using the exception `e`.

7. `return`
   Stops further execution (especially inside a function).

1.4 Part 4 After Logged in - Logging Out

1.4.1 Code Explanation

The `sidebar logout` block provides a user-friendly way to display the current login status and allow the user to log out via a sidebar interface in Streamlit.

1. It checks if both `"logged_in"` and `"username"` exist in `st.session_state`. These conditions confirm the user is logged in.
2. If the user is authenticated:
   - A welcome message is shown in the sidebar using `st.sidebar.success()`, dynamically displaying the current `username`.
3. Below the greeting, a logout button labeled `" Logout"` is rendered using `st.sidebar.button()`.
4. When the logout button is clicked, the `on_click` callback triggers `st.session_state.clear()`, which removes all session variables, effectively logging the user out and resetting the app state.

[ ]:

```python
# --- Sidebar logout ---
if st.session_state.get("logged_in") and st.session_state.get("username"):
    st.sidebar.success(f" Logged in as {st.session_state.username}")  # Welcome message
    st.sidebar.button(
        " Logout",
        on_click=lambda: st.session_state.clear()  # Clear session on logout
    )
```

1.5 Part 5 After Logged in - The Searching Engine

1.5.1 Code Explanation

This section defines the main logic flow of the app, combining user profile input, energy calculations, search functionality, and result visualization.

1. If the user_profile exists in st.session_state, it extracts the user's data fields including gender, age, height, weight, activity_level, and goal.
2. The sidebar displays the extracted profile using st.sidebar.write() with formatted text.
3. Energy needs are calculated using two custom functions:
   - calculate_tee() for Total Energy Expenditure (TEE).
   - calculate_bmr() for Basal Metabolic Rate (BMR). The BMI is also computed using the standard formula.
4. The estimated time and distance required to burn 1/3 of TEE (one meal's worth) are computed via calories_to_exercise_with_distance() and shown in the sidebar per activity.
5. In the main panel, users can enter a food keyword (default is "beef") to search USDA's API.
6. Once the "Find Foods" button is clicked:
   - Step 1: The search_usda_foods() function retrieves a list of fdcIds using the keyword.
   - Step 2: These IDs are passed to fetch_multiple_foods() to retrieve full nutritional data.
   - Step 3: The raw API data is processed into a structured DataFrame using extract_nutrients_df().
7. Step 4: If the dataset contains calorie data, the app estimates how much activity is needed to burn the average calories.
8. Step 5: Using the user profile, the TEE and target macronutrients per meal are calculated. Then score_menu() is called to generate a weighted score for each food based on how closely it matches the user's nutritional targets.
9. The scored results are displayed in a table using st.dataframe() and introduced with a formatted subheader.
10. Finally, a radar chart is rendered for each food using plot_radar_chart(), iterating through each row and displaying the nutrient breakdown visually in two-column layout using st.columns(2).

[ ]:

```
# --- Input Section (for logged-in users only) ---
if "user_profile" in st.session_state:
    profile = st.session_state.user_profile

    # Extract personal info from session state
    gender = profile["gender"]
    age = profile["age"]
    height = profile["height"]
    weight = profile["weight"]
    activity_level = profile["activity_level"]
    goal = profile["goal"]

    # --- Sidebar: Display user profile and metrics ---
    st.sidebar.markdown("###  Your Profile")
    st.sidebar.write(f"** Gender:** {gender}")
    st.sidebar.write(f"** Age:** {age} years")
    st.sidebar.write(f"** Height:** {height} cm")
    st.sidebar.write(f"** Weight:** {weight} kg")
    st.sidebar.write(f"** Activity Level:** {activity_level}")
    st.sidebar.write(f"** Goal:** {goal.replace('_', ' ').title()}")

    # Calculate energy needs and activity equivalents
    tee = calculate_tee(gender, age, height, weight, activity_level)      #␣
↪Total Energy Expenditure
    bmi = weight / ((height / 100) ** 2)                                   #␣
↪Body Mass Index
    burn_data = calories_to_exercise_with_distance(tee / 3, bmi, age)     # Burn␣
↪1 meal worth of kcal

    bmr = calculate_bmr(gender, age, height, weight)                      #␣
↪Basal Metabolic Rate

    # --- Sidebar: Display BMR and TEE results ---
    st.sidebar.markdown("###  Daily Energy Estimates")
    st.sidebar.write(f"** BMR:** **{round(bmr)}** kcal/day")
    st.sidebar.write(f"** TEE:** **{round(tee)}** kcal/day")

    # --- Sidebar: Display burn estimates for 1/3 TEE ---
    st.sidebar.markdown("###  Burn 1 Meal (~ TEE):")
    for activity, stats in burn_data.items():
        st.sidebar.write(f"**{activity}**: {stats['time_min']} min ␣
↪{stats['distance_km']} km")

    # --- Main panel: search bar ---
    st.markdown("###  Search Food by Keyword")
    keyword = st.text_input("Search food keyword", value="beef")  # default =␣
↪"beef"
    submitted = st.button(" Find Foods")

    # --- Search logic begins ---
    if submitted:
        # Step 1: Search fdcIds by keyword
        fdc_ids = search_usda_foods(keyword, API_KEY)

        # Step 2: Fetch nutrient data by fdcIds
        foods = fetch_multiple_foods(fdc_ids, API_KEY)

        # Step 3: Convert to structured dataframe
        df = extract_nutrients_df(foods)
        # Step 4: (Optional) Estimate how much effort needed to burn average food calories
        if "Calories" in df.columns:
            avg_calories = df["Calories"].mean()
            bmi = weight / ((height / 100) ** 2)
            exercise_data = calories_to_exercise_with_distance(avg_calories, bmi, age)

        # Step 5: Score food based on user profile
        tee = calculate_tee(gender, age, height, weight, activity_level)
        targets = compute_target_macros_per_meal(tee)
        scored = score_menu(df, targets, tee, goal)

        # --- Output ranked results ---
        st.subheader(
            f" Top Foods for '{keyword}' (Goal: {goal.replace('_', ' ').title()})"
        )
        st.dataframe(scored)  # Show table with ranking

        # --- Show radar charts for each food item ---
        cols = st.columns(2)  # Two-column layout for radar charts
        for i, (_, row) in enumerate(scored.iterrows()):
            with cols[i % 2]:
                st.markdown(f"####  {row['Food']} - {row['Brand']}")
                plot_radar_chart(row)
```

1.5.2 Further Explanation for Lines of Code that could be Confusing

Line 74 to 75

1. `scored.iterrows()` returns each row in the DataFrame as a `(index, row)` tuple.
2. `enumerate(...)` adds a counter `i` (0, 1, 2, . . .).
3. `_` is used to ignore the original index since it's not needed. The reason it's not needed is because it has no meaning after we sort the data based on nutrient score.
4. `row` contains the actual data from the DataFrame.
5. `cols[i % 2]` alternates between `cols[0]` and `cols[1]` using the modulo operator (`% 2`), placing content in the left and right columns in turn.
6. The `with` block scopes layout content inside the chosen column.