Before You Begin: Prerequisites

1. **Google Account**: You'll need a Google account to use Google Colab.
2. **Hugging Face Account & Token**:
    - You'll need a Hugging Face account to download models like Mistral-7B.
    - Create an Access Token on Hugging Face (go to your Profile -> Settings -> Access Tokens -> New token with "read" or "write" role).
    - In your Colab notebook, you'll need to add this token as a "Secret". Click the "🔑" (Key) icon on the left sidebar, then "+ Add new secret". Name it HF_TOKEN and paste your token value.
3. **Familiarity with Colab**: Basic understanding of running cells in Colab will be helpful.

How to Use This Guide with Your Notebook

Your Colab notebook (Naptick_Task2.ipynb) is already structured in steps. This guide mirrors that structure.

- **Run Cells Sequentially**: It's crucial to run the cells in your notebook in the order they appear. Each step often depends on the successful completion of the previous ones.
- **Monitor Outputs**: Pay attention to the output of each cell. Look for success messages (✅), warnings (⚠️), or errors (❌).
- **GPU Runtime**: Ensure your Colab runtime is set to use a GPU for efficient model operation.
    - Go to **Runtime -> Change runtime type**.
    - Select **T4 GPU** (or other available GPU) from the "Hardware accelerator" dropdown.
    - Click **Save**. If you change the runtime type, you'll likely need to restart the runtime and run all cells from the beginning.

Code Structure & Explanation

Brief Code Write-Up (by Line Numbers):

- **Lines 1-48: Initial Setup & Configuration**
    - Mounts Google Drive (line 4).

    - Installs necessary Python libraries like langchain, transformers, faiss-cpu, gradio, etc. (lines 6-9).
    - Sets up project directories in Google Drive (lines 14-26).
    - Logs into Hugging Face Hub using a Colab secret (HF_TOKEN) (lines 29-37).
    - Checks library versions and GPU availability (lines 40-48).
- **Lines 50-199: Sample Data Generation**
    - **Lines 50-83:** Creates sample_wearable_data.csv with simulated sleep, activity, and heart rate metrics.
    - **Lines 86-96:** Creates sample_chat_history.json with a few conversational turns.
    - **Lines 99-120:** Creates main_user_profile.json with a generic user persona and preferences.
    - **Lines 123-135:** Creates sample_location_data.csv with simulated travel/location events.
    - **Lines 138-199:** Creates .txt files in custom_collection containing general knowledge about sleep (e.g., hygiene, sleep stages).
- **Lines 202-270: Data Loading and Preprocessing**
    - Loads data from the generated files into Langchain Document objects.
    - Wearable (lines 207-218) and Location (lines 240-251) data (CSV) are loaded manually with Pandas and converted to Document objects.
    - Chat History (JSON) is loaded manually (lines 221-231).
    - User Profile (JSON) is loaded using JSONLoader (lines 236-239).
    - Custom Collection text files are loaded using TextLoader (lines 254-258).
    - **Lines 262-270 (Chunking):** Documents from custom_collection (text-heavy) are split into smaller chunks using RecursiveCharacterTextSplitter for better embedding and retrieval.
- **Lines 273-290: Embedding Model Initialization**
    - Initializes a sentence transformer model (sentence-transformers/all-MiniLM-L6-v2) using HuggingFaceEmbeddings for converting text to numerical vectors. Configures it to use GPU if available.
- **Lines 293-341: FAISS Vector Store Creation/Loading**
    - Defines paths for saving FAISS indexes.
    - Iterates through each data collection:
        - If an existing FAISS index is found, it's loaded.
        - Otherwise, a new FAISS index is created from the documents (using the initialized embedding model) and saved to disk. This creates separate vector stores for each data type.
- **Lines 344-414: Retrieval Function Implementation**
    - Defines retrieve_context(query, k_per_store):
        - Searches each FAISS vector store for documents similar to the user's query.
        - Retrieves k_per_store top documents from each (with a special condition for 'location' to retrieve more).
        - Deduplicates retrieved documents based on content.
        - Formats the relevant documents into a single context string for the LLM.
    - Includes a test block (if __name__ == '__main__':) to demonstrate retrieval.

- **Lines 417-470: LLM Initialization and Prompt Template**
  - Loads the Large Language Model (mistralai/Mistral-7B-Instruct-v0.2) using AutoModelForCausalLM and its tokenizer.
  - Applies 4-bit quantization (BitsAndBytesConfig) if a GPU is available to reduce memory usage (lines 424-430).
  - Wraps the LLM and tokenizer in a Hugging Face pipeline for text generation, then in a Langchain HuggingFacePipeline (lines 445-451).
  - Defines a detailed PromptTemplate (lines 454-469) that instructs the LLM on how to behave, how to use the provided context and chat history, and how to format its answer. This prompt is crucial for RAG.
- **Lines 473-513: RAG Chain and Memory Implementation**
  - Initializes ConversationBufferMemory to store chat history.
  - Defines prepare_chain_inputs: a function that takes a user query, retrieves context using retrieve_context, fetches chat history from memory, and prepares all inputs for the prompt template.
  - Constructs the rag_chain using Langchain Expression Language (LCEL):
    1. Passes input through.
    2. Calls prepare_chain_inputs.
    3. Formats the prompt.
    4. Sends to the LLM.
    5. Parses the output to a string.
  - Includes a test block for the chain and memory.
- **Lines 516-613: Gradio User Interface**
  - Defines print_memory_usage for logging resource consumption.
  - Clears conversation memory for a fresh Gradio session.
  - **stream_response_gradio(user_message) (lines 533-590):**
    - Handles a user message for streaming output.
    - Uses TextIteratorStreamer for token-by-token generation from the LLM.
    - Prepares inputs, formats the prompt, and starts LLM generation in a separate thread.
    - Yields text chunks as they are generated.
    - Cleans the final response and saves the conversation to memory.
  - **chat_interface_fn_gradio(user_message, history) (lines 593-599):** The main function Gradio calls, which uses stream_response_gradio to provide streaming output to the UI.
  - Creates the gr.ChatInterface with title, description, examples, and UI element settings (lines 602-610).
  - Launches the Gradio app (chat_app.launch()) (line 613), making it accessible via a local and potentially a public URL.