**Hemant Goyal**  
**Naptick AI Challenge Submission - Task 1**  
**Multi-Collection RAG System with Memory Layer**  
**24/05/25**

**1. Introduction**

This document outlines the implementation of a multi-collection Retrieval Augmented Generation (RAG) system designed as a sleep and wellness AI assistant. The system, "Naptick AI," processes queries by retrieving relevant information from five distinct data collections (wearable data, chat history, user profile, location data, and a custom knowledge base) and uses a Large Language Model (LLM) to generate contextualized answers. It also incorporates a memory layer to maintain conversation context. The project is developed as a Google Colab notebook and features a Gradio web interface for interaction.

The core architecture focuses on generalizing all data sources, treating them as a unified knowledge base without user-specific filtering, simplifying the retrieval for the current single-persona ("Alice") setup.

**2. Project Setup and Prerequisites**

**2.1. Environment**

The project is designed to run in a **Google Colaboratory (Colab)** environment, preferably with a GPU runtime (e.g., T4) for efficient model loading and inference.

**2.2. Required Accounts and Tokens**

* **Google Account:** For accessing Google Colab and Google Drive.
* **Hugging Face Account:** To download the pre-trained language model (Mistral-7B-Instruct-v0.2) and embedding model (all-MiniLM-L6-v2).
* **Hugging Face User Access Token:** This token must be stored as a Colab Secret named HF\_TOKEN for authenticated access to Hugging Face Hub.

**2.3. Google Drive Structure**

The code will create the following directory structure in your Google Drive:

/content/drive/MyDrive/Naptick\_Challenge/

**3. Code Explanation**

**Step 0: Mount Drive and Install Libraries**

* **Purpose:** Sets up the Colab environment.
* **Actions:**
  1. Mounts Google Drive to /content/drive for persistent storage of data and vector stores.
  2. Installs necessary Python libraries using pip:
     + langchain, langchain-community: Core framework for building RAG applications.
     + python-dotenv: (Though not explicitly used for HF token here as Colab secrets are preferred).
     + sentence-transformers: For generating text embeddings.
     + faiss-cpu: For efficient similarity search in vector stores (CPU version, but GPU is used for model inference).
     + gradio: For creating the web UI.
     + tiktoken: Tokenizer used by LangChain.
     + accelerate, bitsandbytes, einops: For optimizing LLM loading and inference, especially with quantization.
     + transformers, torch: For Hugging Face models and PyTorch operations.
     + jq: For JSON processing by JSONLoader.
* **Output:** Confirmation of library installation.

**Step 1: Project Setup and Hugging Face Login**

* **Purpose:** Defines project paths, creates directory structures, logs into Hugging Face, and checks library versions.
* **Actions:**
  1. Defines project\_path, data\_base\_path, and vector\_store\_base\_path on Google Drive.
  2. Creates these directories and subdirectories for each data collection if they don't exist using os.makedirs.
  3. Attempts to log into Hugging Face Hub using the HF\_TOKEN retrieved from Colab secrets.
  4. Prints versions of transformers, torch, and langchain.
  5. Checks for GPU availability and prints the GPU name if found.
* **Output:** Confirmation of folder creation, Hugging Face login status, library versions, and GPU detection.

**Step 2: Create Sample Data (Generalized)**

This step generates dummy data for the five required collections. All data is "generalized," meaning it does not contain specific user\_id fields, simplifying the RAG system for a single-persona context.

* **Step 2a: Wearable Data (sample\_wearable\_data.csv)**
  + Simulates 10 days of wearable data (sleep stages, activity, heart rate) from sources like WHOOP, Fitbit, and Apple Health.
  + Data is stored in a CSV file without a user\_id column.
  + Columns: timestamp, source, metric, value.
* **Step 2b: Chat History (sample\_chat\_history.json)**
  + Creates a sample conversation log between a "user" and "bot."
  + Stored as a JSON list of message objects, each with timestamp, speaker, and text.
* **Step 2c: User Profile (main\_user\_profile.json)**
  + Defines a single user profile (aliased as "Alice") with preferences, goals, and notes.
  + Stored as a JSON object.
* **Step 2d: Location Data (sample\_location\_data.csv)**
  + Simulates a few location check-ins with descriptions.
  + Columns: timestamp, latitude, longitude, place\_description.
* **Step 2e: Custom Collection (Text Files)**
  + Creates several .txt files containing general knowledge about sleep hygiene, deep sleep, REM sleep, caffeine effects, and sleep stages. This forms the system's general knowledge base.
* **Output (for each sub-step):** Confirmation of data file creation and a sample of the generated data.

**Step 3: Load Data from Collections**

* **Purpose:** Loads the generated data into LangChain Document objects, preparing them for embedding and indexing. This step was critically revised to manually load CSV data for better control over page\_content.
* **Actions:**
  1. **Wearable & Location Data (Manual CSV Loading):**
     + Uses pandas.read\_csv to load the CSV files.
     + Iterates through each row, manually constructing a descriptive page\_content string for each Document.
     + Stores all original CSV columns as metadata for each Document, along with collection name and original\_source\_file.
  2. **Chat History (Manual JSON Loading):**
     + Loads the JSON file and iterates through messages, creating a Document for each with page\_content as "speaker: text" and relevant metadata.
  3. **User Profile (JSONLoader):**
     + Uses DirectoryLoader with JSONLoader to load the main\_user\_profile.json.
     + jq\_schema='.' loads the entire JSON object as content, which is then converted to a string.
     + A metadata\_func adds the collection name.
  4. **Custom Collection (TextLoader):**
     + Uses DirectoryLoader with TextLoader to load all .txt files from the custom\_collection directory.
  5. All loaded documents are organized into a dictionary docs\_by\_collection.
* **Output:** Confirmation of document loading from each collection, sample content, metadata, and the total number of documents loaded.

**Step 4: Chunk Documents**

* **Purpose:** Splits large documents (primarily from the custom\_collection) into smaller, manageable chunks for more effective embedding and retrieval.
* **Actions:**
  1. Initializes RecursiveCharacterTextSplitter with a defined chunk\_size (512 characters) and chunk\_overlap (50 characters).
  2. Splits only the documents from the custom\_collection.
  3. Combines the original (unchunked) documents from other collections with the chunked documents from custom\_collection into final\_docs\_for\_embedding.
* **Output:** Confirmation of splitter initialization, chunking time, number of original vs. chunked documents, and total documents ready for embedding.

**Step 5: Initialize Embedding Model**

* **Purpose:** Loads a pre-trained sentence transformer model to convert text documents into numerical vector embeddings.
* **Actions:**
  1. Defines the model name: sentence-transformers/all-MiniLM-L6-v2.
  2. Sets the device to cuda if available, otherwise cpu.
  3. Initializes HuggingFaceEmbeddings with the chosen model, device, and normalize\_embeddings=True (beneficial for similarity search).
  4. Performs a test embedding to confirm functionality.
* **Output:** Confirmation of model configuration, successful loading, and test embedding results (including dimension).

**Step 6: Create and Save/Load FAISS Vector Stores**

* **Purpose:** Creates or loads FAISS vector stores (indexes) for each data collection. FAISS allows for efficient similarity search on the document embeddings. This step now includes explicit deletion of old indexes to ensure fresh data is used.
* **Actions:**
  1. **Delete Old Indexes (Crucial New Sub-step):**
     + Iterates through expected index paths and uses shutil.rmtree to delete any pre-existing FAISS index directories. This forces the system to rebuild indexes with the latest data and Document structures from Step 3.
  2. Defines paths for saving FAISS indexes within the vector\_store\_base\_path.
  3. Iterates through each collection:
     + Uses the appropriate documents (chunked for custom\_collection, original for others).
     + If an index directory does *not* exist (because it was just deleted or is a first run), it creates a new FAISS index using FAISS.from\_documents() with the documents and the initialized embedding model.
     + Saves the newly created index locally to Google Drive using vector\_store.save\_local().
     + If an index *does* exist (e.g., if deletion failed or this sub-step is run partially), it attempts to load it using FAISS.load\_local(). allow\_dangerous\_deserialization=True is used as FAISS stores pickled Python objects.
  4. Stores references to the loaded/created vector stores in loaded\_vector\_stores.
* **Output:** Logs for index deletion attempts, index creation/loading times, paths, and any errors.

**Step 7: Implement Retrieval Function (Generalized)**

* **Purpose:** Defines the retrieve\_context function, which searches the FAISS vector stores for documents relevant to a user's query.
* **Actions:**
  1. The function takes a query and k\_per\_store (number of documents to retrieve per store) as input.
  2. It iterates through each collection in loaded\_vector\_stores.
  3. **Enhanced k for Location:** If the collection is location, current\_k is set to max(k\_per\_store, 5) to ensure more location-specific documents are retrieved.
  4. Performs a similarity search (vector\_store.similarity\_search(query, k=current\_k)) on each vector store.
  5. Adds debugging logs to show the top retrieved documents specifically from the location collection.
  6. Collects all retrieved documents, de-duplicates them based on page content.
  7. Formats the unique documents into a single context string, prepending each with its source collection and filename.
  8. Includes debug logging for the final context of travel/location queries.
* **Output (during tests):** Debug logs showing the retrieval process, retrieved documents, and the final formatted context.

**Step 8: Initialize LLM and Prompt Template**

* **Purpose:** Loads the Large Language Model (LLM) and defines the prompt template that will guide its responses.
* **Actions:**
  1. Defines the LLM model ID: mistralai/Mistral-7B-Instruct-v0.2.
  2. Configures 4-bit quantization (BitsAndBytesConfig) if a GPU is available, to reduce memory usage.
  3. Loads the model using AutoModelForCausalLM.from\_pretrained() with quantization and device\_map="auto".
  4. Loads the corresponding tokenizer using AutoTokenizer.from\_pretrained(). Sets pad\_token if not already defined.
  5. Creates a Hugging Face pipeline for text generation with parameters like max\_new\_tokens, do\_sample, temperature, etc.
  6. Wraps this pipeline in HuggingFacePipeline for LangChain compatibility.
  7. Defines a PromptTemplate with placeholders for context, chat\_history, query, and user\_name ("Alice" by default). The prompt instructs the LLM to answer based strictly on the provided context and history, and to use a specific fallback message if information is not found.
* **Output:** Confirmation of LLM/tokenizer loading, pipeline creation, and prompt template definition.

**Step 9: Implement Memory Layer and RAG Chain (Generalized)**

* **Purpose:** Sets up conversational memory and constructs the main RAG chain using LangChain Expression Language (LCEL).
* **Actions:**
  1. Initializes ConversationBufferMemory to store chat history.
  2. Defines prepare\_chain\_inputs function:
     + Takes the user query.
     + Calls retrieve\_context (from Step 7) to get relevant documents.
     + Loads current chat history from conversation\_memory.
     + Sets user\_name\_for\_prompt (e.g., "Alice").
     + Returns a dictionary of inputs for the prompt template.
  3. Constructs the rag\_chain using LCEL:
     + RunnablePassthrough(): Passes the initial query.
     + .assign(prepared\_inputs=prepare\_chain\_inputs): Calls the input preparation function.
     + Selects the prepared\_inputs.
     + Pipes into prompt\_template.
     + Pipes into the llm\_pipeline.
     + Pipes into StrOutputParser() to get a string output.
* **Output:** Confirmation of memory initialization and RAG chain creation. Test queries demonstrate the chain's functionality and memory updates.

**Step 10: Gradio Interface (Using Generalized RAG)**

* **Purpose:** Creates a user-friendly web interface for interacting with the RAG chatbot.
* **Actions:**
  1. Imports gradio and other necessary modules.
  2. Defines print\_memory\_usage helper function for monitoring resource consumption.
  3. Clears conversation\_memory for a fresh Gradio session.
  4. Defines stream\_response\_gradio(user\_message):
     + This is the core function handling a user request.
     + Initializes TextIteratorStreamer for streaming LLM responses.
     + Calls prepare\_chain\_inputs (with k\_per\_store=2 for Gradio for a balance of speed and context).
     + Formats the prompt.
     + Sets up generation arguments for the text\_gen\_pipeline, including the streamer.
     + Starts the LLM generation in a separate Thread.
     + Iterates through the streamer, yielding each new text chunk to Gradio for progressive UI updates.
     + Concatenates chunks to full\_response.
     + After streaming, joins the thread, processes the full\_response (e.g., truncating potential hallucinations like repeated "User:" or "Assistant:" turns).
     + Saves the user message and final bot response to conversation\_memory.
     + Includes extensive logging and error handling.
  5. Defines chat\_interface\_fn\_gradio(user\_message, history):
     + The function that Gradio's ChatInterface calls.
     + It calls stream\_response\_gradio and yields its output to update the Gradio chat UI.
  6. Creates gr.ChatInterface with the defined function, title, description, example questions, and UI component configurations. Button arguments are set to None to prevent potential TypeErrors observed in earlier development iterations.
  7. Launches the Gradio app using chat\_app.launch(debug=True, share=True).
* **Output:** A running Gradio web interface, accessible via a local URL and potentially a public share URL. Console logs will show request processing, context retrieval, and memory usage.

**4. How to Run the Notebook**

1. **Open in Google Colab:** Upload or open the .ipynb file in Google Colab.
2. **Set Hugging Face Token:**
   * In Colab, click on the "Key" icon (Secrets) in the left sidebar.
   * Add a new secret named HF\_TOKEN.
   * Paste your Hugging Face User Access Token (with read permissions) as the value.
   * Ensure "Notebook access" is toggled on for this secret.
3. **Select Runtime:**
   * Go to "Runtime" -> "Change runtime type."
   * Select a **GPU** accelerator (e.g., "T4").
4. **Run Cells Sequentially:**
   * Execute each cell in the notebook from top to bottom.
   * **First Run:** Allow time for libraries to install and models to download (LLM and embedding model). This can take several minutes.
   * **Data and Index Generation:** Steps 2 and 6 will create files and folders in your Google Drive under /content/drive/MyDrive/Naptick\_Challenge/. The first time Step 6 runs after the index deletion sub-step, it will build all FAISS indexes. Subsequent runs of Step 6 (if the deletion sub-step is skipped or fails and indexes exist) will attempt to load existing indexes.
5. **Interact with Gradio:**
   * Once Step 10 is executed, a Gradio interface will launch. Click on the local URL (usually ends with gradio.live if share=True or a localhost URL) provided in the output to open the chatbot UI in a new browser tab.
   * Ask questions using the input textbox. Examples are provided in the Gradio interface.

**5. Key Technologies and Components**

* **Programming Language:** Python
* **Environment:** Google Colaboratory
* **Core Framework:** LangChain
* **LLM:** mistralai/Mistral-7B-Instruct-v0.2 (via Hugging Face Transformers)
* **Embedding Model:** sentence-transformers/all-MiniLM-L6-v2
* **Vector Store:** FAISS (Facebook AI Similarity Search)
* **UI:** Gradio
* **Key Libraries:** transformers, torch, pandas, numpy, bitsandbytes, accelerate.

**6. Observations and Potential Future Enhancements**

* **Generalized Retrieval:** The current system treats all data as general knowledge. For a multi-user scenario, the original user-specific filtering logic in retrieve\_context would need to be re-instated and refined.
* **Context Window:** The Mistral-7B model has a context window. Very long conversations or extremely large retrieved contexts might exceed this, leading to truncation or loss of older information.
* **Retrieval Tuning:** The k\_per\_store values and the page\_content formatting are crucial for effective retrieval. Further experimentation could optimize these. For instance, using different k values for different query types or collections might be beneficial.
* **Advanced Chunking:** More sophisticated chunking strategies (e.g., semantic chunking) could be explored.
* **Re-ranking:** A re-ranking step after initial retrieval (e.g., using a cross-encoder) could improve the relevance of the final documents sent to the LLM.
* **Error Handling:** While robust error handling is in place, specific edge cases in data or queries might require further refinement.
* **Scalability:** For a production system with many users and larger datasets, more scalable vector database solutions (e.g., Pinecone, Weaviate, managed FAISS) would be considered.