Project - RAG LLM

Aim

To develop a structured pipeline for retrieving historical information about India from Wikipedia, processing it into meaningful embeddings, storing it in a vector database, and building a Retrieval-Augmented Generation (RAG) chatbot that answers user queries accurately using FAISS and the Gemini API.

Acceptance Criteria

1. **Data Scraping**
   - Successfully extract and store historical Wikipedia pages related to Indian history in JSON format.
   - Ensure that only relevant Wikipedia links are collected while avoiding irrelevant pages like "Category" and "File" links.
   - Implement rate-limiting mechanisms to prevent getting blocked by Wikipedia.
2. **Chunking, Embedding, and Storing in FAISS**
   - Successfully load the **BERT model** and generate embeddings for historical text.
   - Implement **semantic chunking** to divide long texts into manageable segments.
   - Store all embeddings and metadata efficiently in the FAISS vector database.
   - Ensure the FAISS index is properly saved and can be reloaded for future queries.
3. **Query Processing and Answer Generation**
   - Convert user queries into embeddings and retrieve the most relevant historical documents from FAISS.
   - Generate responses using the **Gemini API**, incorporating retrieved historical content.
   - Ensure that responses are **factually accurate** and **contextually relevant** to user queries.
   - Provide a user-friendly **Streamlit interface** for seamless interaction.

1 - Scrapping the Data - History of India

Source - Wikipedia

Purpose

- Automate the extraction of historical Wikipedia pages starting from the 𝕎 "History of India" page.
- Collect **titles and main content** from relevant Wikipedia pages.
- Store the extracted information in a **structured JSON format** for future analysis.

Technologies Used

- **Python**: Main programming language.
- **Requests**: To send HTTP requests and fetch webpage content.
- **BeautifulSoup**: For parsing and extracting data from Wikipedia pages.
- **tqdm**: To display progress while scraping multiple pages.
- **JSON**: To store structured data in a readable format.

Code Workflow

The script follows a **systematic process** to extract and store data:

Step 1: Fetch Wikipedia Links

- Start from the 𝕎 "History of India" Wikipedia page.
- Identify and collect **valid Wikipedia links** related to history.
- **Filter out** irrelevant links such as files, categories, and special pages.

Step 2: Scrape Wikipedia Content

- For each valid link:
  - Extract the **page title**.
  - Extract the **main content** (excluding navigation and citations).
  - Store the data in a structured format.

Step 3: Store Data

- Convert the collected data into a **structured JSON file** (`wikipedia_history.json`).
- Ensure that Unicode characters are stored correctly.

Step 4: Prevent Rate Limiting

- Introduce **random delays (1-3 seconds)** between requests to avoid getting blocked by Wikipedia.

Code Explanation

```
WIKI_BASE_URL = "https://en.wikipedia.org"
```

- Defines the base Wikipedia URL.

Extract Valid Wikipedia Links

```
# Function to extract valid Wikipedia links
def get_wiki_links(url, visited):
    """Extracts valid Wikipedia history-related links from a given page."""
    response = requests.get(url)
    soup = BeautifulSoup(response.text, "html.parser")
    links = set()

    for link in soup.find_all("a", href=True):
        href = link["href"]
        if href.startswith("/wiki/") and ":" not in href and href not in visited:
            full_url = WIKI_BASE_URL + href
            links.add(full_url)

    return links
```

- Fetches all links from the given Wikipedia page.
- Ensures only **valid** Wikipedia article links are considered.
- Filters out **special pages** like `Category:` , `File:` , `Help:` .

Scrape Wikipedia Content

```
# Function to scrape content from a Wikipedia page
def scrape_wikipedia_page(url):
    """Scrapes the title and main content of a Wikipedia page."""
    response = requests.get(url)
    soup = BeautifulSoup(response.text, "html.parser")

    # Extract the title
    title = soup.find("h1").text.strip()

    # Extract the main content
    content = []
    for p in soup.find_all("p"):
        text = p.text.strip()
        if text:
            content.append(text)

    return {"title": title, "content": " ".join(content)}
```

- Extracts the **title** and **main textual content** of a Wikipedia page.
- Ignores **irrelevant sections** like tables, sidebars, and references.

Main Execution Flow

```

# Starting from the Wikipedia page - history of India
start_url = "https://en.wikipedia.org/wiki/History_of_India"
visited_pages = set()
scraped_data = []

# Get initial links from the main History of India page
wiki_links = get_wiki_links(start_url, visited_pages)

print(f"✅ Found {len(wiki_links)} potential pages to scrape.")

```

- Starts from the **History of India** page.
- Collects valid Wikipedia links.

Scrape Multiple Pages

```python
# Scrape pages up to 1000
for url in tqdm(wiki_links, desc="Scraping Wikipedia", total=1000):
    if len(scraped_data) >= 1000:
        break

    if url not in visited_pages:
        try:
            page_data = scrape_wikipedia_page(url)
            scraped_data.append(page_data)
            visited_pages.add(url)
        except Exception as e:
            print(f" Error scraping {url}: {e}")

        # Random sleep to avoid getting blocked
        time.sleep(random.uniform(1, 3))
```

- Scrapes **up to 1000** Wikipedia pages.
- **Handles exceptions** gracefully (e.g., network failures).
- **Random delays** prevent getting blocked by Wikipedia.

```
shtlp_0165@SHTLP0165:~/Desktop/Project-RAG-LLM$ /bin/python3 /home/shtlp_0165/Desktop/Project-RAG-LLM/scrapping.py
✅ Found 1909 potential pages to scrape.
Scraping Wikipedia: 100%|                                  | 1000/1000 [1:15:44<00:00,  4.54s/it]
✅ Successfully scraped 1000 Wikipedia pages!
shtlp_0165@SHTLP0165:~/Desktop/Project-RAG-LLM$
```

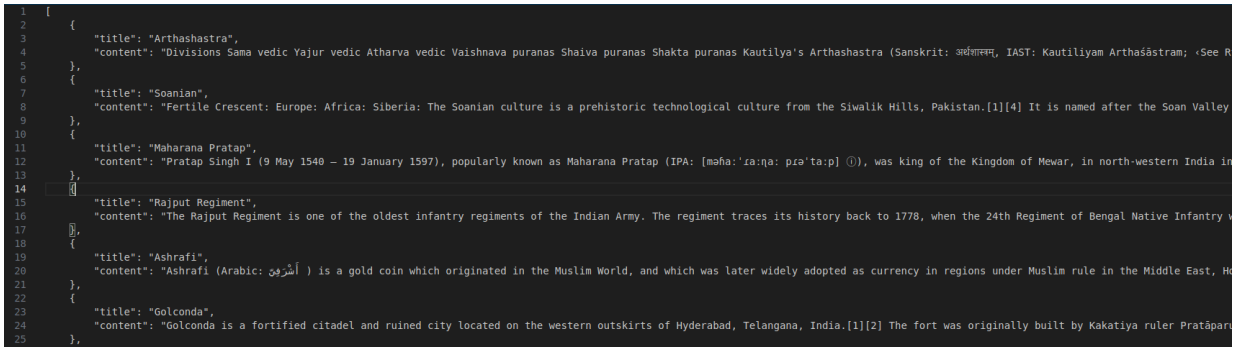
Save Data in JSON

```python
# Saving the scraped data as JSON
with open("wikipedia_history.json", "w", encoding="utf-8") as f:
    json.dump(scraped_data, f, indent=4, ensure_ascii=False)

print(f"✅ Successfully scraped {len(scraped_data)} Wikipedia pages!")
```

- Stores the scraped data in a **structured JSON format**.
- Ensures **Unicode compatibility**.

Output



```
[
    {
        "title": "Arthashastra",
        "content": "Divisions Sama vedic Yajur vedic Atharva vedic Vaishnava puranas Shaiva puranas Shakta puranas Kautilya's Arthashastra (Sanskrit: अर्थशास्त्र, IAST: Kautiliyam Arthaśāstram; ‹See R
    },
    {
        "title": "Soanian",
        "content": "Fertile Crescent: Europe: Africa: Siberia: The Soanian culture is a prehistoric technological culture from the Siwalik Hills, Pakistan.[1][4] It is named after the Soan Valley
    },
    {
        "title": "Maharana Pratap",
        "content": "Pratap Singh I (9 May 1540 – 19 January 1597), popularly known as Maharana Pratap (IPA: [məɦaːˈrɑːɳaː prəˈtaːp] ⓘ), was king of the Kingdom of Mewar, in north-western India in
    },
    {
        "title": "Rajput Regiment",
        "content": "The Rajput Regiment is one of the oldest infantry regiments of the Indian Army. The regiment traces its history back to 1778, when the 24th Regiment of Bengal Native Infantry
    },
    {
        "title": "Ashrafi",
        "content": "Ashrafi (Arabic: أَشْرَفِيّ ) is a gold coin which originated in the Muslim World, and which was later widely adopted as currency in regions under Muslim rule in the Middle East, H
    },
    {
        "title": "Golconda",
        "content": "Golconda is a fortified citadel and ruined city located on the western outskirts of Hyderabad, Telangana, India.[1][2] The fort was originally built by Kakatiya ruler Pratāpar
    },
```

2- Chunking, Embedding and Storing in Vector DB

Hugging Face BERT Embeddings

BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based model developed by Google that has significantly advanced natural language understanding tasks. By capturing context from both directions, BERT produces embeddings that encapsulate the semantic meaning of text, making it ideal for tasks like semantic search.

FAISS Vector Database

FAISS is a library developed by Facebook AI that enables efficient similarity search and clustering of dense vectors. It is particularly useful for large-scale data, offering rapid approximate nearest neighbor searches.

System Components and Workflow

The system comprises several key components:

a. Importing Libraries

Essential libraries such as `json`, `faiss`, `numpy`, `os`, `torch`, and classes from `langchain` are imported to handle various functionalities, including data processing, embedding generation, and vector indexing.

b. Loading the BERT Model and Tokenizer

The BERT model and its corresponding tokenizer are loaded using Hugging Face's `transformers` library. The specific model used is `sentence-transformers/all-MiniLM-L6-v2`, a variant optimized for producing sentence embeddings efficiently.

chunking_embedding_vectorize.py > process_json

```python
'''Hugging Face BERT Embedding'''
import json
import faiss
import numpy as np
import os
from abc import ABC, abstractmethod
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.schema import Document
from dotenv import load_dotenv
from transformers import AutoModel, AutoTokenizer
import torch

# Load Hugging Face Model
model_name = "sentence-transformers/all-MiniLM-L6-v2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)
```

Generating Embeddings

A function `get_embedding` is defined to convert input text into embeddings. The text is tokenized and passed through the BERT model, and mean pooling is applied to obtain a fixed-size vector representation.

```python
def get_embedding(text):
    tokens = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)
    with torch.no_grad():
        embeddings = model(**tokens).last_hidden_state.mean(dim=1)  # Mean pooling
    return embeddings.squeeze().numpy().tolist()
```

Text Chunking

To manage lengthy documents, a `SemanticChunker` class is implemented using LangChain's `RecursiveCharacterTextSplitter`. This class divides text into manageable chunks, ensuring that each segment is within the model's processing limits.

```python
# Semantic Chunker (LangChain)
class SemanticChunker(BaseChunker):
    def __init__(self, chunk_size=512, chunk_overlap=50):
        self.text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=chunk_size, chunk_overlap=chunk_overlap, separators=["\n\n", "\n", " ", ""]
        )

    def chunk(self, text: str):
        return self.text_splitter.split_text(text)
```

FAISS Vector Database Management

The `FAISSVectorDB` class manages the indexing and retrieval of document embeddings. It handles adding new documents, saving and loading the index, and storing metadata.

```python
# FAISS Vector Database Class
class FAISSVectorDB:
    def __init__(self, index_file="faiss_index.bin", metadata_file="documents_metadata.json"):
        self.index_file = index_file
        self.metadata_file = metadata_file
        self.index = None
        self.docs = []

        self.load_index()
        self.load_metadata()

    def add_documents(self, docs):
        embeddings = np.array([get_embedding(doc.page_content) for doc in docs], dtype=np.float32)

        if self.index is None:
            self.index = faiss.IndexFlatL2(embeddings.shape[1])  # Initialize FAISS

        faiss.normalize_L2(embeddings)
        self.index.add(embeddings)
        self.docs.extend(docs)

        self.save_index()
        self.save_metadata()

    def save_index(self):
        if self.index is not None:
            faiss.write_index(self.index, self.index_file)

    def load_index(self):
        if os.path.exists(self.index_file):
            self.index = faiss.read_index(self.index_file)
            print("✅ FAISS index loaded.")
        else:
            print("⚠ No FAISS index found. A new one will be created.")

    def save_metadata(self):
        print(f"⚡ Saving {len(self.docs)} metadata entries to {self.metadata_file}...")  # Debugging print
        with open(self.metadata_file, "w", encoding="utf-8") as f:
            json.dump([{"title": doc.metadata["title"], "content": doc.page_content} for doc in self.docs], f, indent=4)
        print("✅ Metadata file saved successfully!")


    def load_metadata(self):
        if os.path.exists(self.metadata_file):
            with open(self.metadata_file, "r", encoding="utf-8") as f:
                self.docs = [Document(page_content=doc["content"], metadata={"title": doc["title"]}) for doc in json.load(f)]
            print("✅ Document metadata loaded.")
        else:
            print("⚠ No metadata found. It will be created.")
```

f. Processing JSON Data

The `process_json` function reads a JSON file containing documents, chunks the content, and adds the resulting segments to the FAISS index.

```python
# Function to Process JSON Data
def process_json(json_file, chunker, vector_db):
    with open(json_file, "r", encoding="utf-8") as file:
        data = json.load(file)

    docs = []
    for entry in data:
        title = entry["title"]
        content = entry["content"]
        chunks = chunker.chunk(content)

        for chunk in chunks:
            doc = Document(page_content=chunk, metadata={"title": title})
            docs.append(doc)

    print(f"✅ Adding {len(docs)} documents to FAISS")
    vector_db.add_documents(docs)

    print("⚡ Manually saving metadata...")
    vector_db.save_metadata()
```

Execution Flow

The main execution block of the script performs the following steps:

- **Initialization**: It initializes the `SemanticChunker` with a chunk size of 512 and an overlap of 50 characters. Simultaneously, it sets up the `FAISSVectorDB` with specified file paths for the FAISS index and metadata storage.
- **Model Verification**: A test embedding is generated using the `get_embedding` function with the input "Hello, test query" to ensure that the Hugging Face BERT model is functioning correctly.
- **Index Verification**: The script checks if the FAISS index is already built by verifying the existence of the index file and the number of vectors (`ntotal`) in the index.
- **Index Construction**: If the FAISS index is absent or empty, the script processes the `wikipedia_history.json` file. It uses the `SemanticChunker` to divide the content into manageable chunks and adds these chunks to the FAISS index.
- **Metadata Saving**: After processing, the script manually saves the metadata to ensure synchronization between the index and its associated metadata.
- **Completion Message**: Finally, the script outputs the total number of vectors present in the FAISS index, indicating the completion of the

indexing process.

This structured approach ensures that the text data is efficiently processed, embedded, and indexed, facilitating rapid similarity searches using the FAISS library.

3- Answering the Query:

Retrieval-Augmented Generation (RAG) Chatbot Overview

Retrieval-Augmented Generation (RAG) is a technique that enhances Large Language Models (LLMs) by integrating an information retrieval system. Instead of relying solely on pre-trained knowledge, the chatbot retrieves relevant documents from a database and uses them to generate accurate, fact-based responses. This approach reduces misinformation and improves contextual accuracy.

System Components

1. Hugging Face BERT Embeddings

The chatbot uses the **sentence-transformers/all-MiniLM-L6-v2** model from Hugging Face to generate numerical representations (embeddings) of text. These embeddings capture the semantic meaning of the text, enabling effective similarity searches.

Code Snippet: Initializing the BERT Model

Query.py > ...

```python
import os
import json
import faiss
import numpy as np
import streamlit as st
from google import genai
from dotenv import load_dotenv
from transformers import AutoTokenizer, AutoModel
import torch


# Initializing Hugging Face BERT Model for embeddings
model_name = "sentence-transformers/all-MiniLM-L6-v2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)
```

Function to Generate Embeddings

```python
# Function to generate embeddings using Hugging Face BERT
def get_embedding(text):
    tokens = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=512)
    with torch.no_grad():
        embeddings = model(**tokens).last_hidden_state.mean(dim=1)
    return embeddings.squeeze().numpy()
```

This function converts input text into a vector representation using BERT.

2. FAISS Vector Database

FAISS is a library for fast searching of high-dimensional embeddings. The chatbot stores document embeddings in a FAISS index and uses it to find the most relevant documents based on similarity.

Code Snippet: Loading the FAISS Index

```python
# Loading  FAISS index
index_file = "faiss_index.bin"
if not os.path.exists(index_file):
    raise FileNotFoundError(f"FAISS index file '{index_file}' not found.")
index = faiss.read_index(index_file)   #index(1000,384)
```

3. Processing User Queries

When a user submits a query, it is converted into an embedding and compared with stored document embeddings to find the most relevant information.

Code Snippet: Query Processing

```python
# Function to process user query
def process_query(query, top_k=5):
    query_embedding = np.array([get_embedding(query)], dtype=np.float32)
    faiss.normalize_L2(query_embedding)

    # Search FAISS index
    '''Distance will be a list of distance of vectors with the query vector
        indices are index of the those documents
        '''
    distances, indices = index.search(query_embedding, top_k)

    # Retrieving relevant documents
    results = []
    for idx in indices[0]:
        if idx < len(documents_metadata):   #If Index exists
            doc = documents_metadata[idx]
            results.append(doc)

    return results

```

This function:

- Converts the query into an embedding
- Normalizes it for better comparison
- Searches for the top-k closest matches in the FAISS index
- Retrieves metadata of matching documents

4. Generating Responses with Gemini API

The chatbot sends the retrieved document content along with the user's query to **Google's Gemini API**, which generates a response.

Code Snippet: Generating AI Response

```python
# Function to generate response using Gemini API
def generate_response(query):
    relevant_docs = process_query(query, top_k=5)

    if not relevant_docs:
        return "No relevant information found in the knowledge base."

    #Preparing context for LLM
    context = "\n".join([doc["content"][:500] for doc in relevant_docs])

    #Prompt to the Gemini
    prompt = f"Based on the following information:\n{context}\n\nAnswer this question:\n{query}"

    response = client.models.generate_content(
    model="gemini-2.0-flash", contents=prompt
    )

    return response.text.strip()

```

This function:

- Retrieves the most relevant documents
- Prepares a **prompt** with retrieved context
- Sends the prompt to **Gemini API**
- Returns the generated response

5. Streamlit User Interface

The chatbot interface is built using **Streamlit**, providing real-time interaction.

Code Snippet: Setting Up the Streamlit Interface

```python
# Streamlit UI
st.set_page_config(page_title="RAG Chatbot by Nilesh Mehra", page_icon=":speech_balloon:")

st.title("RAG Chatbot")
st.caption("Developed by Nilesh Mehra")


if "chat_history" not in st.session_state:
    st.session_state.chat_history = []

# Display chat history
for chat in st.session_state.chat_history:
    with st.chat_message(chat["role"]):
        st.markdown(chat["content"])

# User input
if "end_chat" not in st.session_state or not st.session_state.end_chat:
    user_query = st.chat_input("Enter your query:")
    if user_query:
        # Display user message
        st.chat_message("user").markdown(user_query)
        # Generate and display assistant response
        response = generate_response(user_query)
        st.chat_message("assistant").markdown(response)
        # Update chat history
        st.session_state.chat_history.append({"role": "user", "content": user_query})
        st.session_state.chat_history.append({"role": "assistant", "content": response})

# End chat button
if st.button("End Chat"):
    st.session_state.end_chat = True
    st.write("Thank you for using the chatbot. This service is developed by Nilesh Mehra.")

```

This part of the code:

- Displays chat history
- Takes user input
- Calls the **generate_response()** function to fetch the response
- Updates the chat history dynamically

Execution Flow

1. The FAISS index and document metadata are loaded.
2. The chatbot interface is set up using Streamlit.
3. The user enters a query.
4. The query is converted into an embedding.
5. The FAISS index is searched for similar documents.
6. The most relevant documents are used as context for the **Gemini API**.
7. The chatbot generates a response and displays it to the user.
8. The chat history is updated.
9. The user can continue chatting or end the session.

Code Flow Diagram

Below is a flow diagram illustrating the interaction between the various components of the chatbot system:

```
User
  |
  ▼
Streamlit Interface
  |
  ▼
User Query
  |
  ▼
Generate Query Embedding (BERT)
  |
  ▼
Search FAISS Index
  |
  ▼
Retrieve Relevant Documents
  |
  ▼
Construct Prompt with Retrieved Documents
  |
  ▼
Send Prompt to Gemini API
  |
  ▼
Receive Generated Response
  |
  ▼
Display Response in Streamlit Interface
  |
  └─> Update Chat History
```

Outcome

The project successfully automates Wikipedia data extraction, processes text using BERT embeddings, and stores it in a FAISS vector database for efficient retrieval. The RAG-based chatbot enhances responses with relevant historical context, leveraging the Gemini API. With a user-friendly Streamlit interface, users can quickly access accurate historical information in a scalable and efficient manner.