RAG System

This documentation provides an overview of the Retrieval-Augmented Generation (RAG) model, including its setup, usage, and key functions. The RAG model combines retrieval-based and generative approaches to produce accurate and contextually relevant responses to user queries.

Setup

Requirements

- Python 3.8+
- Required Python packages:
  - selenium
  - langchain
  - faiss
  - huggingface_hub

Installation

Install the required Python packages using pip:

```
pip install selenium langchain faiss huggingface_hub
```

Usage

Initialization

The RAG model requires data collection, which is achieved through web scraping. The scraped data is processed and stored in a text file.

1. **Scrape Data**
   The `data_collection` function scrapes data from a specified website using Selenium and saves it in `meta.txt`.
2. **Define API Keys**
   Set the required API keys for Hugging Face and Groq:

```
os.environ['HUGGINGFACEHUB_API_TOKEN'] = "123456" groq_api_key = "123456"
```

Define Embedding Model

Load the embedding model from Hugging Face:

huggingface_embeddings = HuggingFaceHubEmbeddings(
model="BAAI/bge-small-en-v1.5")

Load LLM Model

Initialize the language model from Groq:

```
llm = ChatGroq(groq_api_key=groq_api_key,
model_name="Llama3-8b-8192")
```

Set Prompt Template

Define the prompt template for the model:

```
prompt = ChatPromptTemplate.from_template(""" Answer the questions
based on the provided context only. Please provide the most accurate
response based on the question <context> {context} <context>
Questions:{input} """)
```

Query Processing

1.Load Data
Load the contents of `meta.txt`:

```
loader = TextLoader("meta.txt")
docs = loader.load()
```

2.Text Splitting and Vector Embeddings

Split the text into chunks and create vector embeddings:

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=200)
final_documents = text_splitter.split_documents(docs)
vectors = FAISS.from_documents(final_documents,
huggingface_embeddings)
```

3.Run Query

Initialize the document chain and retriever, then process the query:

```
document_chain = create_stuff_documents_chain(llm, prompt)
retriever = vectors.as_retriever()
retrieval_chain = create_retrieval_chain(retriever, document_chain)
```

```
prompt1 = ""
while prompt1 != 'byebye':
    prompt1 = input("Enter Your Question From Documents: ")
    response = retrieval_chain.invoke({'input': prompt1})
    print("Response:", response['answer'])
print("Thank You")
```

Working

When you run `home.py`, the following steps are executed:

1. **Data Collection**:
    - The `data_collection()` function is called.
    - This function uses Selenium to scrape data from a predefined website (`https://botpenguin.com/`).
    - The scraped content is stored in a text file named `meta.txt`.
2. **Text Loading**:
    - The content of `meta.txt` is loaded using `TextLoader`.
    - The loaded text is then processed into vector embeddings using the Hugging Face embedding model.
3. **Query Processing**:
    - The user can ask questions through the command line.
    - The text is split into chunks and converted into vector embeddings.
    - A retrieval chain is created to fetch relevant documents based on the user's query.
    - The model generates a response based on the retrieved documents and the query.
    - The response is displayed on the command line.

Embedding Model

The embedding model used in this project is the Hugging Face BAAI/bge-small-en-v1.5. This model converts text into high-dimensional vectors that capture semantic information, allowing the system to understand and retrieve relevant content based on the user's query. These embeddings are essential for comparing and matching user queries with the stored document chunks.

LLM Model

The Language Model (LLM) used is ChatGroq, specifically the "Llama3-8b-8192" model. This model is known for its efficient and accurate generation capabilities. It's used to generate responses to user queries by leveraging the context provided by the retrieved

documents. The model is optimized for faster response times, making it suitable for real-time query handling.

Vector Data

The vector data is managed using FAISS (Facebook AI Similarity Search). FAISS is a library for efficient similarity search and clustering of dense vectors. It stores the vector embeddings of the document chunks and enables rapid retrieval of the most relevant vectors in response to a query. This retrieval process ensures that the most contextually appropriate pieces of information are passed to the LLM for generating a precise and accurate answer.

Conclusion

The Retrieval-Augmented Generation (RAG) model implemented in this project demonstrates a powerful approach to handling user queries with both retrieval and generative capabilities. By combining advanced techniques such as web scraping, vector embeddings, and state-of-the-art language models, the system achieves robust performance in understanding and responding to user queries.