Retrieval-Augmented Generation (RAG) for Local Data Processing

Project Overview

This repository focuses on creating a **local Retrieval-Augmented Generation (RAG)** system designed to understand your queries through a **Large Language Model (LLM)**, search your local database, and semantically retrieve relevant information. By integrating the **Ollama** platform, this project enables an advanced AI system similar to **ChatGPT**, but it operates entirely on your **local machine** without requiring an **internet** connection.

The local RAG system leverages the LLM's ability to **comprehend** the nuances of your **query**, enabling it to search through your dataset and extract **semantically** related information. Unlike a traditional database query, which requires **precise keywords** or structured queries (such as SQL), the RAG system allows for natural language input. This means that instead of returning **exact keyword matches**, the RAG system can **interpret** the **context** and **intent** behind your query, pulling information that is **semantically relevant** even if it **doesn NOT match exact keywords**.

For example, in a **regular database query**, asking for "customer complaints about delivery delays" might only return entries explicitly tagged with "delivery delay." In contrast, a **RAG-based system** can understand **synonyms**, **variations**, or **implicit** mentions of the issue, such as "shipment took too long" or "late delivery." This makes it significantly more powerful when searching through unstructured or loosely structured data like **customer reviews**, **patient records**, or **emails**.

By performing all tasks **locally**, the system guarantees **data privacy** and ensures compliance with **confidentiality** regulations, making it ideal for handling **sensitive information** such as **patient records** or **proprietary customer feedback**. Additionally, the combination of retrieval and generation offers robust **data analysis**, **report generation**, and **decision support** capabilities.

Integrating a **RAG system** into your database infrastructure can dramatically enhance the ability to extract actionable insights from large datasets. Instead of relying on rigid, predefined query patterns, a RAG system empowers users to engage with their data in a more **intuitive** and **context-aware** manner. This enables more **flexible exploration** of data, more accurate **information retrieval**, and the ability to generate **coherent explanations** for complex queries—all while maintaining full **control** over data workflows and privacy.

This project not only provides a powerful tool for managing and interpreting your own data but also serves as a learning resource for building customized RAG systems. Users can explore how the system semantically searches their databases and generates context-aware responses, enhancing the capability of traditional database management.

Key Concepts

1. Retrieval-Augmented Generation (RAG)

RAG enhances large language models by retrieving relevant information from external data sources, such as documents, databases, or a vector database. The system represents local documents as embeddings (vectors), which allows the model to find information based on the input query.

RAG is particularly effective when:

- The language model lacks sufficient context or knowledge about a particular subject.
- The data being queried is large, but only certain relevant sections are needed.
- The queries should be answered using specific or recent data from local files.

2. Vector Database

A **vector database** is a specialized database designed to store and retrieve high-dimensional vectors, also known as embeddings. In this RAG setup, documents are converted into embeddings (vector representations) and stored in the vector database. When a query is made, it is also converted into a vector, and the vector database quickly retrieves the closest matching document embeddings based on semantic similarity.

Vector databases are essential for scaling retrieval tasks, as they allow efficient search across large datasets by finding the most relevant vectors in a multi-dimensional space. Here, Ollama leverages **embeddings** to represent document meaning and stores them in an internal vector database for efficient retrieval.

3. Embeddings

Embeddings are vector representations of text that capture the semantic meaning of words or documents. The system uses Ollama's `mxbai-embed-large` model to create high-quality embeddings for the local documents. These embeddings are stored in a vector database, enabling fast and accurate retrieval based on user queries.

Embeddings serve as the backbone of the RAG system. By transforming text into a numerical format, the system can quickly compare and retrieve the most relevant documents or portions of data.

4. Local LLMs with Ollama

Ollama is an open-source platform that allows the running of large language models (LLMs) directly on your machine, without needing cloud-based services. This local execution ensures data privacy and minimizes latency while still maintaining the powerful capabilities of models like **llama3**.

Setup

The following steps will help you set up the Local RAG system on your laptop. This setup is designed to be lightweight and straightforward, making it ideal for educational purposes or prototyping.

Step 1: Clone the Repository

First, clone this repository to your local machine:

```
git clone https://github.com/memari-majid/local_rag.git
```

Go to the repository downloaded in your local machine:

```
cd local_rag
```

Step 2: Install Python Dependencies

Install the required Python packages by running the following command:

```
pip install -r requirements.txt
```

These dependencies include tools for managing language models, handling documents, and generating embeddings.

Step 3: Install Ollama

Ollama is the core platform for running the local language models. Download and install it from the official website:

Download Ollama

Once installed, Ollama will allow access to a variety of optimized language models for local use.

Step 4: Pull the Required Models

You will need to download the specific models that power the RAG system.

llama Models and Memory Requirements

llama models, especially **llama 3**, are advanced large language models developed by **Meta** for **Natural Language Processing (NLP)** tasks. These models are **open-weight** and **optimized** for various tasks, such as text generation, summarization, and question-answering. **llama 3**, the latest iteration, offers enhanced capabilities in terms of speed, **efficiency**, and accuracy compared to its predecessors. However, each model size requires different amounts of memory (RAM), depending on the number of parameters. **Ollama** can help alleviate some of the hardware requirements associated with running large models like llama 3 by providing **optimized** infrastructure and tools to manage models more efficiently.

- **llama 3 - 7B**: The **smallest** model with 7 billion parameters. It is **efficient** and requires around **16 GB** of GPU memory to run smoothly. Suitable for tasks requiring less computational power while still maintaining strong language understanding capabilities.

- **llama 3 - 13B**: A **mid-size** model with 13 billion parameters. It strikes a **balance** between performance and efficiency and requires approximately **24 GB** of GPU memory, making it suitable for more complex tasks without the need for high-end hardware.

- **llama 3 - 34B**: A **larger** model with 34 billion parameters. It delivers higher **accuracy** and performance but requires more computational resources, typically needing around **48 GB** of GPU memory to operate effectively.

- **llama 3 - 70B**: The **largest** model with 70 billion parameters. It provides the **highest** accuracy and performance but requires significant hardware **resources**, including **80 GB** or more of GPU memory, making it ideal for the most demanding language generation tasks.

When selecting a llama model, it is important to consider the available memory and hardware capabilities, as larger models require more memory but deliver better performance for complex tasks.

Use the following commands to pull the necessary models:

```
# Pull llama 3 - 7B
ollama pull llama3:7b
```

Generating High-Quality Embeddings from Documents

The `mxbai-embed-large` model is designed to generate high-quality embeddings from textual documents. Embeddings are vector representations of text that capture the **semantic** meaning of words or sentences. These embeddings are essential for tasks like similarity search, document clustering, and retrieval-augmented generation (RAG) systems.

By leveraging a large embedding model like `mxbai-embed-large`, you can ensure that your documents are encoded into dense vectors that preserve their **semantic** richness and **contextual** information, which improves the performance of downstream tasks such as search, recommendation, or summarization.

To pull the `mxbai-embed-large` model using **ollama**, use the following command:

mxbai-embed-large for generating high-quality embeddings from documents:

```
ollama pull mxbai-embed-large
```

These models will be stored locally, allowing the entire system to run offline.

Step 5: Upload Your Documents

The `upload.py` script allows users to upload documents in **PDF**, **TXT**, and **JSON** formats and append their contents to a file called `vault.txt`. This tool is designed to handle documents by converting them into chunks of text, making them ready for further processing.

Key Features of `upload.py`:

- **PDF to Text Conversion**: Converts PDF files to text and normalizes the text by removing unnecessary whitespace. The text is split into chunks (up to 1000 characters) and appended to `vault.txt`, with each chunk on a new line.
- **TXT File Upload**: Reads text from `.txt` files, normalizes whitespace, and splits the content into 1000-character chunks. The text is then appended to `vault.txt`, similar to the PDF functionality.
- **JSON File Upload**: Parses JSON files and flattens the data into a single string. It then processes the content by normalizing whitespace and splitting it into chunks before appending it to `vault.txt`.

How the Script Works:

1. **User Interface**: A simple GUI is provided using `tkinter`, where buttons allow users to select and upload their files.
2. **PDF Handling**: When a user selects a PDF file, the script extracts the text from each page, normalizes it, and splits it into chunks of up to 1000 characters.
3. **Text File Handling**: When a `.txt` file is uploaded, the text is similarly cleaned and split into chunks of up to 1000 characters before being added to `vault.txt`.
4. **JSON File Handling**: For JSON files, the script flattens the entire structure into a single text string, normalizes whitespace, and then splits the content into chunks before appending to `vault.txt`.

Example Usage:

To upload files using the script, simply run the script and select the file format you wish to upload:

```
python upload.py
```

This will open a GUI where you can choose a PDF, TXT, or JSON file. The selected file will be processed, and its content will be appended to `vault.txt` in chunks.

Adding New Document Types: If you want to extend the functionality of upload.py to support additional file types, you can modify the script by adding new functions similar to upload_txtfile or upload_jsonfile.

This script makes it easy to gather documents from various formats into a central text file, ready for processing or analysis.

Step 6: Query Your Documents

The `localrag.py` script provides a local environment for running Retrieval-Augmented Generation (RAG) using a vault of documents. This script allows you to query a set of local documents and retrieve relevant context for answering questions. It integrates with **Ollama** to generate embeddings and retrieve context based on user input. Additionally, it utilizes **PyTorch** to handle similarity calculations and embeddings.

Key Features of `localrag.py`:

- **Document Retrieval**: The script reads from a local text file (`vault.txt`), generates embeddings for the contents, and uses cosine similarity to retrieve relevant text based on the user's query.
- **Ollama Integration**: The script uses **Ollama's** embedding and completion models, such as `mxbai-embed-large` for generating embeddings and `llama3` for answering user queries based on the context.
- **Query Rewriting**: When a user asks a question, the script can rewrite the query by referencing recent conversation history, improving the query to retrieve better results without changing its original intent.

How the Script Works:

1. **Loading Documents**: The script reads a text file named `vault.txt` containing the documents you want to use for context retrieval. Each line of the file is treated as a separate chunk of text, and

embeddings are generated for these chunks using the **mxbai-embed-large** model.

2. **Generating Embeddings**: Once the vault is loaded, the script creates embeddings for each chunk using Ollama's embedding model. These embeddings are stored as tensors and used for similarity searches.

3. **User Interaction**:

   - The user is prompted to input a query. If context from the vault is relevant, the script retrieves the top matching chunks based on cosine similarity.
   - If there is sufficient conversation history, the script rewrites the user's query using the most recent context.
   - The query, along with the relevant context, is passed to **Ollama's language model** (e.g., `llama3`) for generating a response.

4. **Conversation Loop**: The script enters an interactive loop where the user can continually input queries and receive responses. The conversation history is updated with each interaction to improve context.

Example Usage:

To run the `localrag.py` script, use the following command in your terminal:

```
python localrag.py
```

This will initiate the conversation loop. The system will use the llama3 model by default for processing your queries. You can specify a different model by passing the --model argument.

Vault File: Ensure that your `vault.txt` is properly formatted, with each document or chunk of text on a new line. The script will generate embeddings for each line and use them to find the most relevant context for your queries.

Example Commands:

- **Ask a Query:** Type a question and the script will retrieve the most relevant content from vault.txt, rewriting your query if needed, and provide a response.

- **Quit the Script:** To exit the conversation, simply type quit.

Configuration:

- **Ollama API:** The script is configured to interact with an Ollama API running locally on http://localhost:11434/v1. Ensure that the API is running before executing the script.

- **Embedding Model:** By default, the script uses the mxbai-embed-large model to generate document embeddings.

This script is highly flexible for building local RAG systems, enabling users to query a custom set of documents and receive intelligent, context-aware responses.

Using `localrag_no_rewrite.py` for Retrieval-Augmented Generation (RAG)

The `localrag_no_rewrite.py` script is a simplified version of a Retrieval-Augmented Generation (RAG) system that allows you to query a set of local documents and retrieve relevant context without rewriting the user's query. This script uses **Ollama** for generating embeddings and answering user queries by pulling relevant context from a local "vault" of documents.

Key Features of `localrag_no_rewrite.py`:

- **No Query Rewriting**: Unlike other RAG systems, this script does not rewrite the user's query before retrieving relevant context. Instead, it directly uses the user's input for context retrieval.
- **Document Retrieval**: The script reads from a local text file (`vault.txt`), generates embeddings for the contents using **Ollama's** embedding model, and retrieves the most relevant chunks based on the user's query.
- **Ollama Integration**: The script utilizes **Ollama's** API for both embedding generation (e.g., `mxbai-embed-large`) and text generation (e.g., `llama3` or other specified models).
- **Cosine Similarity Matching**: The system uses cosine similarity to match the user's input with the most relevant chunks of text from the vault, providing highly relevant context for answering queries.

How the Script Works:

1. **Loading Documents**: The script reads a file named `vault.txt` containing chunks of documents or text lines. Each line in the file is treated as a document, and embeddings are generated for each chunk using the **mxbai-embed-large** model.
2. **Generating Embeddings**: Embeddings are generated for the vault content using **Ollama's** embedding model. These embeddings are stored as tensors and used to compute cosine similarity for relevance matching.
3. **User Interaction**:
   - The user provides a query. The system retrieves the top relevant chunks of text from the vault based on cosine similarity.
   - The relevant context is concatenated with the user’s input and sent to **Ollama's** language model (e.g., `llama3`) to generate a response.
4. **Conversation Loop**: The script continuously prompts the user for input, retrieves context, and provides responses in a loop. The conversation history is maintained to allow continuity in the dialogue.

Example Usage:

To run the `localrag_no_rewrite.py` script, use the following command in your terminal:

```
python localrag_no_rewrite.py
```

This will initiate the conversation loop. By default, the system uses the llama3 model, but you can specify another model using the --model argument.

Vault File: Ensure that your vault.txt is properly formatted, with each document or chunk of text on a new line. The script will generate embeddings for each line, which are used to match the user's query with relevant content.

Example Commands:

- **Ask a Query:** Type your question and the system will retrieve relevant content from vault.txt and provide a response.
- **Quit the Script:** Type quit to exit the conversation loop.

Configuration:

- **Ollama API:** The script interacts with an Ollama API running locally on http://localhost:11434/v1. Ensure that the API is running before executing the script.
- **Embedding Model**: The script uses the mxbai-embed-large model for generating document embeddings by default.

This script provides a simple, yet powerful tool for querying local documents and generating responses using RAG techniques without modifying the user's original input.