RAG System Details till 04 Oct 2024

1. Overview of Files

Main Files:

- `main.py`:
  - The entry point for the application, which leverages **Gradio** to create a user-friendly interface for training and testing the RAG system.
  - It provides two primary functionalities:
    - **Train Model**: Allows users to upload PDF files and processes them into a Chroma database.
    - **Test Model**: Allows users to input queries, retrieve relevant documents, and generate responses.
- `query_data.py`:
  - Core logic for querying the Chroma vector database and interacting with the **Ollama** model for generation.
  - This file includes functions for:
    - **Query Expansion**: Enhances user queries by generating related terms using the Ollama model.
    - **Document Retrieval**: Retrieves relevant documents from the vector database.
    - **Contextual Compression**: Filters out irrelevant content from retrieved documents before passing them to the language model.
- `populate_database.py`:
  - Manages the ingestion of PDF documents, processes them into chunks, and stores them in a **Chroma vector database**.
  - It includes functionality for:
    - **Loading PDFs**: Extracts text from PDF files.
    - **Splitting Documents**: Breaks large documents into smaller chunks for more efficient retrieval.
    - **Adding to Chroma**: Embeds the document chunks and adds them to the Chroma vector store.
- `get_embedding_function.py`:
  - Defines the embedding model (Ollama's **nomic-embed-text**) that is used in both the document ingestion and retrieval processes.
- `evaluate.py`:
  - Handles the evaluation of the **RAG system's** retrieval and generation performance.
  - It compares generated responses and retrieved document sources against a **ground truth** dataset using metrics such as **Recall@K**, **Mean Average Precision (MAP)**, and **Exact Match**.

2. Detailed Technical Breakdown of Functions

`main.py`

1. **`reset_database()`**:
   a. Clears the Chroma vector database, deleting all existing data.
   b. Calls the `clear_database()` function from `populate_database.py`.
2. **`train_model(file_paths)`**:
   a. Accepts file paths (uploaded via Gradio) and moves them to the DATA_PATH directory.
   b. It then processes these files by calling the following functions from `populate_database.py`:
      i. **`load_documents()`**: Loads the PDF documents.
      ii. **`split_documents()`**: Splits the documents into manageable chunks.
      iii. **`add_to_chroma()`**: Adds these chunks to the Chroma database.
3. **`test_model(query)`**:
   a. Accepts a user query and passes it to `query_rag()` from `query_data.py`.
   b. Displays the generated response and the sources of the information retrieved from the Chroma database.
4. **Gradio Interface**:
   a. **`train_interface`**: A Gradio interface for uploading files and training the model.
   b. **`test_interface`**: A Gradio interface for testing queries against the trained model.

`query_data.py`

1. **`main()`**:
   a. A test function that runs a predefined query ("When did the defect occur?") through the RAG system.
2. **`query_rag(query_text: str)`**:
   a. **Core Functionality**:
      i. **Query Expansion**: Expands the query to generate synonyms using the Ollama model phi3.5
      ii. **Document Retrieval**: Retrieves relevant document chunks from the Chroma vector store.
      iii. **Contextual Compression**: Compresses and filters irrelevant content using the `EmbeddingsFilter`.
      iv. **Response Generation**: Sends the compressed context and query to the Ollama language model (mistral 7B) to generate a response.
      v. **Result**: Returns the response and the sources used.
3. **`expand_query(query_text: str)`**:
   a. Uses the Ollama model (`phi3.5`) to generate synonyms and related terms based on the input query.
   b. Expands the original query by appending these synonyms to improve retrieval accuracy.

`populate_database.py`

1. **`main(data_path)`**:
   a. The main entry point for populating the Chroma database with document embeddings.

b. It checks if the database needs to be reset (via command-line arguments) and processes the documents accordingly.

2. **`load_documents(data_path)`**:
   a. Loads all PDF files from the specified `data_path`.
   b. Calls the `preprocess_pdf()` function to extract text from each PDF file.
3. **`preprocess_pdf(file_path)`**:
   a. Extracts text from a PDF file using the `fitz` library (PyMuPDF).
   b. Cleans and structures the extracted text into specific sections (e.g., Basic Information, Defect Analysis, Defect Resolution).
4. **`split_documents(documents)`**:
   a. Splits each document into smaller chunks using the `RecursiveCharacterTextSplitter` from LangChain.
   b. Ensures chunks overlap slightly to maintain context between chunks.
5. **`add_to_chroma(chunks)`**:
   a. Embeds each document chunk using the embedding function (Ollama's **nomic-embed-text**).
   b. Adds the embedded chunks to the Chroma vector store.
   c. Uses `calculate_chunk_ids()` to ensure each chunk has a unique ID.
6. **`calculate_chunk_ids(chunks)`**:
   a. Generates unique chunk IDs for each document chunk based on its source and chunk index.
   b. Ensures that document chunks are uniquely identifiable within the Chroma vector store.
7. **`clear_database()`**:
   a. Deletes the entire Chroma vector store by removing the `CHROMA_PATH` directory.

`get_embedding_function.py`

1. **`get_embedding_function()`**:
   a. Initializes and returns the **OllamaEmbeddings** model (`nomic-embed-text`).
   b. This embedding function is used in:
      i. **Document Storage**: Embedding document chunks when adding them to the Chroma vector store.
      ii. **Querying**: Embedding queries to retrieve relevant documents from the vector store.

`evaluate.py`

1. **`evaluate_rag_model(ground_truth)`**:
   a. Evaluates the RAG system by comparing generated responses and retrieved sources with a predefined **ground truth dataset**.
   b. For each query in the ground truth, it calculates:
      i. **Recall@K**: How many relevant documents were retrieved within the top K results.
      ii. **Mean Average Precision (MAP)**: The precision of retrieved documents averaged over the query's relevant results.
      iii. **Exact Match**: Whether the generated response exactly matches the expected answer.
2. **`recall_at_k_score(retrieved_sources, true_sources, k=5)`**:

a. Calculates **Recall at K**: How many of the true relevant documents are found in the top K retrieved results.

3. **`mean_average_precision(retrieved_sources, true_sources)`**:
   a. Calculates the **Mean Average Precision (MAP)** of the retrieved documents compared to the true relevant documents.
4. **`exact_match_score(model_answer, true_answer)`**:
   a. Checks if the generated model answer exactly matches the true expected answer from the ground truth.

3. Summary of File Interactions

Core Flow:

1. **Training the Model (via Gradio Interface)**:
   a. **`main.py`** handles the Gradio interface.
   b. When files are uploaded via Gradio, they are processed by:
      i. **`load_documents()`**: Loads PDFs from the `data` directory.
      ii. **`split_documents()`**: Splits the documents into smaller chunks.
      iii. **`add_to_chroma()`**: Embeds the document chunks and stores them in the Chroma vector store.
2. **Querying the Model**:
   a. **`query_rag()`** (from `query_data.py`) is called when a query is submitted through the Gradio interface.
   b. This function:
      i. Expands the query using the Ollama model.
      ii. Retrieves relevant document chunks from the Chroma database.
      iii. Compresses the context (removing irrelevant data).
      iv. Generates a response using the Ollama model (`phi3.5`).
   c. The results (response and sources) are returned and displayed in the Gradio interface.
3. **Evaluation**:
   a. **`evaluate_rag_model()`** compares the RAG system's outputs to a ground truth dataset using metrics like Recall@K, MAP, and Exact Match.

4. Data Paths and Usage

Chroma Vector Store Path:

- **`CHROMA_PATH`** is set to `"chroma"` and used for storing document embeddings.

Data Directory:

- **`DATA_PATH`** is set to `"data"`, and this directory is used for storing uploaded PDF files.

Mistral 7B might be considered better than Llama 2 and Llama 3 for using Retrieval-Augmented Generation (RAG)

1. **Training Data**: Mistral 7B was trained on a larger and more diverse dataset compared to Llama 2 and Llama 3. This could lead to better performance on a wider range of tasks.
2. **Instruction Fine-tuning**: Mistral 7B was specifically fine-tuned on instruction following tasks, which could make it better at following instructions and generating more coherent and contextually appropriate responses. Llama 2 and Llama 3 were not explicitly trained for instruction following.
3. **Context Length**: Mistral 7B has a longer context length than Llama 2 and Llama 3. This means it can handle longer inputs and generate longer outputs, which could be beneficial for RAG applications where the context might be quite long.
4. **Efficiency**: Mistral 7B is known for its efficiency. It uses a technique called **sliding window attention**, which allows it to process longer sequences more efficiently than models that use full attention. This could make it more suitable for real-time applications.