RAG Project Documentation

RAG-LLM Q&A System - Ayush Vishwakarma

Overview

This project implements a Retrieval-Augmented Generation (RAG) system integrated with a Language Model (LLM) to answer questions about historical events, science, and mathematics. It scrapes content from predefined URLs, indexes the data using FAISS, and uses an LLM (Ollama) to generate answers. The system includes a Streamlit-based UI for user interaction and evaluation scripts to measure performance.

Key Features

- Scrapes content from Wikipedia, NASA, National Geographic, History.com, and Britannica.
- Uses `BAAI/bge-large-en-v1.5` for generating embeddings and OLLAMA-3b for response LLMs.
- Stores embeddings in a FAISS vector store for efficient retrieval.
- Provides a Streamlit-based UI for asking questions.
- Generating nearly 1000 Test Cases(Q and A) for better evaluation.
- Includes evaluation scripts to measure performance using metrics like cosine similarity, BLEU, and ROUGE..

Project Flow

The system follows a clear sequence of operations to process data, answer questions, and evaluate performance:

1. **Scraping**:
    - The system starts by scraping content from a predefined list of URLs using *scraper.py*.
    - URLs cover wide range of topics in history (e.g., Civil Rights Movement), science (e.g., Photosynthesis), and mathematics (e.g., Algebra).

- The scraped content is extracted as paragraphs from web pages and stored in scraped_data.txt.

2. **Data Processing and Embedding**:
   - The scraped paragraphs are cleaned and processed in *data_processing.py*.
   - The `BAAI/bge-large-en-v1.5` model generates embeddings for each paragraph, converting text into numerical vectors.
3. **Indexing**:
   - The embeddings and corresponding texts are stored in a FAISS vector store using *vector_store.py*.
   - The FAISS index (`faiss_index.bin`) and texts (`texts.pkl`) are saved for future use, enabling efficient similarity search.
4. **Retrieval and Generation (RAG Pipeline)**:
   - When a user submits a question via the Streamlit UI, the question is embedded using the same model.
   - The RAG pipeline (*rag_pipeline.py*) retrieves the most relevant paragraphs from the FAISS index using similarity search.
   - The retrieved paragraphs are passed to the LLM (*llm.py*) to generate a coherent answer.
5. **User Interaction**:
   - The Streamlit UI (*main.py*) displays the generated answer to the user.
   - Users can ask new questions, and the process repeats from step 4.
6. **Test Case Generation**:
   - Test cases are manually generated using Gemini to evaluate the system's performance.
   - Each test case includes a question, its expected answer, and the context (paragraph) from the scraped data used to generate the answer.
   - A total of 1000(nearly) test cases are created, covering various topics from the scraped content

topics from the scraped content.

7. **Evaluation**:
    - The system uses the generated test cases to evaluate performance with *test_rag.py*.
    - Metrics like cosine similarity, BLEU, and ROUGE are computed to assess answer quality.
    - Results are logged and can be extracted for analysis.

File Explanations

Below is a brief explanation of the key files and their roles in the project flow:

main.py

- **Purpose**: Entry point for the application, responsible for initializing the RAG pipeline and running the Streamlit UI.
- **Key Operations**:
    - Defines a list of URLs to scrape.
    - Initializes the embedding model (`BAAI/bge-large-en-v1.5`), vector store, and LLM.
    - Checks for existing FAISS index and texts; if not found, triggers scraping and indexing.
    - Provides a Streamlit UI where users can input questions and view answers.
- **Dependencies**:
    - Imports `scraper.py` , `data_processing.py` , `vector_store.py` , `rag_pipeline.py` , `llm.py` .

scraper.py

- **Purpose**: Handles web scraping to extract content from URLs.
- **Key Operations**:
    - Uses `requests` and `BeautifulSoup` to fetch and parse web pages.

    - Extracts paragraphs by trying multiple HTML tags (`<p>` , `<div>` , `<article>`).
    - Includes retry logic with `tenacity` to handle network failures.
- **Role in Flow**: Provides the raw text data (paragraphs) that will be embedded and indexed.

data_processing.py

- **Purpose**: Processes scraped data and prepares it for indexing.
- **Key Operations**:
    - Sequentially scrapes URLs using the `WebScraper` class from `scraper.py` .
    - Cleans and filters the scraped paragraphs (e.g., removes empty or short texts).
    - Generates embeddings for the paragraphs using the embedding model.
    - Stores the embeddings and texts in the vector store.
- **Role in Flow**: Bridges scraping and indexing by converting raw text into embeddings.

vector_store.py

- **Purpose**: Manages the FAISS vector store for storing and retrieving embeddings.
- **Key Operations**:
    - Implements a `FAISSVectorStore` class to store embeddings and corresponding texts.
    - Provides methods to store, query, save, and load the FAISS index.
    - Includes a dimension check to prevent mismatches between query and index embeddings.
    - Supports Maximal Marginal Relevance (MMR) for diverse retrieval.
- **Role in Flow**: Enables efficient retrieval of relevant paragraphs during the RAG process.

rag_pipeline.py

- **Purpose**: Implements the core RAG pipeline to process user questions and generate answers.
- **Key Operations**:
  - Embeds the user's question using the embedding model.
  - Queries the vector store to retrieve the top-k relevant paragraphs.
  - Formats a prompt with the retrieved paragraphs and passes it to the LLM.
  - Returns the LLM-generated answer.
- **Role in Flow**: Combines retrieval and generation to produce answers from indexed data.

llm.py

- **Purpose**: Interfaces with the Ollama LLM to generate answers.
- **Key Operations**:
  - Defines an `OllamaLLM` class (or similar) to interact with the Ollama model.
  - Takes a prompt (question + retrieved contexts) and generates a response.
- **Role in Flow**: Generates the final answer based on the retrieved contexts.

test_rag.py

- **Purpose**: Evaluates the system's performance on a set of test cases.
- **Key Operations**:
  - Loads a dataset of 1000 test cases (questions and expected answers).
  - Processes each question through the RAG pipeline and compares the generated answer to the expected answer.
  - Computes metrics: cosine similarity, BLEU, and ROUGE.
  - Logs results, including response times and pass/fail status (pass if

exact match or cosine similarity > 0.7).

- **Role in Flow**: Provides a way to measure and validate the system's accuracy and performance.

requirements.txt

- **Purpose**: Lists the Python dependencies required for the project.
- **Contents**:
  - `requests`, `beautifulsoup4`, `tenacity`: For scraping.
  - `sentence-transformers`, `faiss-cpu`, `numpy`, `torch`: For embedding and indexing.
  - `streamlit`: For the UI.
  - `evaluate`, `scikit-learn`, `pytest`: For evaluation and testing.
- **Role in Flow**: Ensures all necessary libraries are installed to run the project.

Setup Instructions

Prerequisites

- Python 3.8 or higher
- Virtual environment (recommended)

Installation

1. Clone the repository:

```shell
git clone https://github.com/your-repo/rag-llm-qa-system.git
cd rag-llm-qa-system

```

2. Create and activate a virtual environment:

```shell
```

```
python -m venv rag_env
source rag_env/bin/activate  # On Windows:
rag_env\Scripts\activate

```

3. Install dependencies:

```shell
pip install -r requirements.txt

```

Initial Setup

- Run the main script to scrape data and build the FAISS index:

```shell
streamlit run main.py

```

- The script will scrape content from predefined URLs, generate embeddings, and save the index to **faiss_index.bin** and **texts.pkl**.

Usage

Running the Application

1. Start the Streamlit app:

```shell
streamlit run main.py

```

2. Open your browser and navigate to `http://localhost:8501`.
3. Enter a question in the text box (e.g., "What is photosynthesis?") and click "Submit".

Example Questions

- What is the history of the Civil Rights Movement?
- How does photosynthesis work?
- What are the key events of the Space Race?

Evaluation

Test Case Generation

To evaluate the system comprehensively, test cases were manually created using Gemini, an AI model, to ensure a diverse and relevant set of questions using the scraped data. Each test case consists of the following components:

- **Question**: A question related to the scraped content (e.g., "What is photosynthesis?").
- **Answer**: The expected answer, generated by Gemini based on the context (e.g., "Photosynthesis is the process by which green plants use sunlight to convert carbon dioxide and water into glucose and oxygen.").
- **Context**: The relevant paragraph(s) from the scraped data that Gemini used to generate the answer (e.g., a paragraph from the National Geographic page on photosynthesis).

Process

1. Gemini was prompted with topics covered by the scraped URLs (e.g., history, science, mathematics).
2. For each topic, Gemini generated questions that a user might ask, ensuring alignment with the scraped content.
3. Gemini then provided the expected answer and identified the specific context (paragraph) from the scraped data that supports the answer.
4. A total of 1000 test cases were created and stored in a dataset ('*all_test_cases.py*') for evaluation.

Example Test Case

- **Question**: What is the significance of the Fall of the Berlin Wall?
- **Answer**: The Fall of the Berlin Wall in 1989 symbolized the end of the Cold War and the division between East and West Germany, leading to German reunification in 1990.
- **Context**: "The Berlin Wall, constructed in 1961, separated East and West Berlin until its fall on November 9, 1989, marking a pivotal moment in the decline of Soviet influence in Eastern Europe." (from Wikipedia)

This manual generation ensured that test cases were both relevant to the scraped content and diverse in scope, covering various domains and question types.

Running Tests

Evaluate the system's performance on the 1000 test cases:

```shell
pytest test_rag.py -v

```

Metrics

- **Cosine Similarity**: Measures embedding similarity between expected and generated answers.
- **BLEU Score**: Evaluates word overlap.
- **ROUGE-1 Score**: Measures unigram overlap.
- **Pass Criteria**: A test case passes if there's an exact match or cosine similarity > 0.7.

Logs

- Logs are generated at the INFO level and can be viewed in the terminal when running the application or tests stored in '*rag_project.log*'.
- Q and A evaluation logs for each test case is stored in

'extracted_scores.json'.

- Terminal logs while running test(*test_rag.py*) are stored in *'terminal_logs.txt'*

Future Improvements

- Add support for dynamic content scraping (e.g., using a headless browser).
- Introduce concurrent scraping to reduce indexing time.
- Enhance the UI with chat history and more interactive features.