Retrieval-Augmented Generation (RAG) Application Documentation

Author: Gaurav Bharatavalli Rangaswamy
Date: 02/22/2025

1. Introduction

This documentation describes a Retrieval-Augmented Generation (RAG) system designed to provide intelligent, context-aware responses to user queries based on an uploaded reference document (PDF/HTML) or a URL. The main objective is to enhance language model outputs by grounding them in factual content retrieved from relevant documents, ultimately giving more accurate and context-specific answers.

The project meets the requirements outlined in the Shyftlabs AI Engineer Take Home Assignment:

- Serves via a webserver (using FastAPI).
- Supports streaming responses to the user.
- Implements both semantic and keyword search.
- Provides an endpoint to upload documents (PDFs or HTML websites).

2. Project Overview

2.1 What is Retrieval-Augmented Generation?

Retrieval-Augmented Generation pairs a Large Language Model (LLM) with a knowledge base or set of documents. When a user asks a question, the system:

1. Retrieves relevant chunks of text from the documents using search mechanisms (keyword search, semantic search, or a combination).
2. Provides these chunks as context to the language model to ground the response and improve factual correctness.

2.2 High-Level Workflow

1. **Upload Documents**: The user can upload a PDF or an HTML file (or provide a URL). The text is extracted, chunked, and embedded.
2. **Query**: The user submits a query.

3. **Search**: The system performs both keyword and semantic search over the stored chunks.
4. **Contextualization**: The retrieved chunks are combined and sent alongside the user query to a Large Language Model (Google's Generative AI, "gemini-pro" in this example).
5. **Response**: The LLM returns a response, optionally streamed back to the user in real time.

3. Application Architecture

Below is an architectural overview of the major components:

1. **FastAPI Web Server**
    - Hosts endpoints for uploading documents, retrieving status, querying, and streaming responses.
    - Manages state (document chunks, metadata) and user interactions.
2. **Document Processing Pipeline**
    - Parses PDF or HTML files to extract text.
    - Splits the extracted text into overlapping chunks for better context retention.
3. **Vector Embedding & Storage**
    - Uses the Sentence Transformers (specifically the `all-MiniLM-L6-v2` model) to generate embeddings for each chunk.
    - Keeps these embeddings in memory (along with the original chunk text).
    - Maintains a simple, persistent store of this data via a pickle file to avoid re-uploading the same document repeatedly.
4. **Search Mechanisms**
    - **Keyword Search**: Checks for direct substring matches of the query within the chunk text.
    - **Semantic Search**: Computes cosine similarity between the query embedding and the stored embeddings, then ranks chunks by relevance.
5. **LLM Integration (Google Generative AI "gemini-pro")**
    - The top relevant chunks from the search stage are combined into a prompt, along with instructions on how to answer (e.g., summary vs. comparison).
    - The LLM is called to produce a final answer, which references the provided context.
6. **Frontend/Chat Interface**
    - A minimal HTML/JS client (`chat.html`) that allows file drag-and-drop or URL input for uploads.
    - Streams the answer back to the user as the LLM generates it.

4. Implementation Details

Below is a closer look at each critical element in the `app.py` FastAPI application and associated files.

4.1 Requirements

A summary of key dependencies from `requirements.txt`:

- **fastapi**: High-performance Python web framework for building APIs.
- **uvicorn**: ASGI server for serving FastAPI.
- **numpy**: Fundamental package for numerical computing and vector operations.
- **PyPDF2**: Library for extracting text from PDF files.
- **beautifulsoup4**: HTML parsing and scraping utility.
- **httpx**: Async HTTP client for Python, used for fetching external URLs.
- **sentence-transformers**: Provides the `all-MiniLM-L6-v2` model for creating text embeddings.
- **scikit-learn**: Used primarily for its `cosine_similarity` function (though also included is a custom function in code).
- **google-generativeai**: Python client for Google's Generative AI models (like `gemini-pro`).

4.2 Document Upload and Parsing

Endpoints:

1. **`POST /upload`**
    - Accepts a file (PDF or HTML) and reads it into memory.
    - If PDF, uses `PyPDF2.PdfReader` to extract text page by page.
    - If HTML, uses `BeautifulSoup` to parse the document, extract `<title>`, meta descriptions, main content, and any special sections.
    - Stores basic metadata (title, type, source filename) to be referenced later.
2. **`POST /upload_url`**
    - Accepts a URL and, using `httpx`, fetches the webpage content.
    - Performs the same HTML parsing described above.
    - Stores metadata similarly (title, source = URL, etc.).

Both endpoints clear any existing state before uploading a new document, ensuring that only one document is indexed at a time.

4.3 Text Chunking

After extracting text, the system splits it into overlapping chunks. The relevant function:

```
def chunk_text(text: str, max_words: int = 150, overlap: int = 30) ->
list:

    ...
```

- Chunks are typically 150 words long with a 30-word overlap to preserve some context across chunk boundaries.
- Overlapping helps the LLM see the continuity of ideas from one chunk to another.

4.4 Embedding Generation

For each chunk, the application generates a vector embedding using:

```
embedding_model = SentenceTransformer('all-MiniLM-L6-v2')
```

This embedding is stored in memory alongside the original chunk text. A snippet from the code:

```
for chunk in chunks:

    embedding = get_embedding(chunk)  # uses the model to encode the
text

    document_chunks.append({"text": chunk, "embedding": embedding})
```

4.5 Document State Persistence

To avoid reprocessing after a server restart, the application:

- Saves the current state (chunks, metadata) to a local pickle file (`document_state.pkl`) via `save_document_state()`.
- Loads the state when the server restarts if the file is fresh (within the last 24 hours).

4.6 Search and Retrieval

When the user queries:

1. **Keyword Search**
   Searches for the lowercase query string within each chunk's text. If found, that chunk is included as a match.
2. **Semantic Search**
   - Uses the same embedding pipeline on the user query.
   - Computes cosine similarity with each chunk's embedding.
   - Ranks chunks by their combined score: primarily the semantic similarity, plus a smaller weighting for term overlap (the ratio of matching words between the query and chunk text).
   - Filters and returns the top matches (default `top_k = 7`).
3. **Result Combination**
   - The application merges keyword and semantic search results into one set to ensure coverage of both direct matches and semantically similar content.

4.7 Prompt Construction for the LLM

The combined chunk text is appended to the query in a specialized prompt that includes:

- Instructions on how to answer (e.g., if the query looks like a comparison or summary).
- Analysis of the query type (greeting, listing, "how to," summary, etc.).
- Document metadata (optional).

The final prompt is structured to encourage the LLM to use only the retrieved context, maintain factual accuracy, and provide clear citations or disclaimers if something is missing.

4.8 LLM Integration and Streaming

Two main endpoints handle user queries:

1. **`GET /query?q=...`**
   - Builds the prompt, sends it to the LLM, and returns the final response in one piece.
2. **`GET /stream_query?q=...`**
   - Streams the LLM response line by line so the user can see partial outputs in real time.

Under the hood:

```
model = genai.GenerativeModel("gemini-pro")

response = model.generate_content(prompt)
```

- The application then processes the returned text and streams it to the client.

4.9 Frontend (chat.html)

- A simple interface with drag-and-drop for PDF/HTML files.
- A URL input field for webpage ingestion.
- A chat area that displays messages in reverse chronological order (newest at the top).
- Real-time streaming of the LLM response.

Once a document is successfully uploaded, the user can ask questions. Each question triggers a request to the server, which returns the answer in real time.

5. Key Technical Features

1. **Overlapping Chunk Strategy**
    - Helps ensure minimal context loss between chunks.
2. **Hybrid Search (Semantic + Keyword)**
    - Improves recall by catching both exact keyword hits and semantically related content.
3. **In-Memory + On-Disk State Management**
    - Speeds up repeated queries without reprocessing the entire file each time, provided the server hasn't been idle more than 24 hours (configurable).
4. **Prompt Engineering**
    - Dynamically constructs a prompt with instructions based on query intent.
5. **Streaming Response**
    - Allows partial text rendering to the user for a more interactive chat experience.
6. **CORS Configuration**
    - Allows cross-origin requests from any domain, making it easy to serve the frontend from a different host or port.

6. Performance and Optimization

- **Efficient Embedding Model**: The `all-MiniLM-L6-v2` model is relatively lightweight but still provides strong semantic embedding performance.
- **Chunk-Level Granularity**: Minimizes the amount of text passed to the LLM at once, reducing token usage and response latency.
- **Caching and State Persistence**: Speeds up repeated queries, only loading embeddings from a pickle file if the server restarts.
- **Adjustable Parameters**:
    - `top_k` in semantic search can be tuned for narrower or broader retrieval.

    - The chunk size (`max_words`) and overlap can be optimized to balance context recall with performance.

7. Security Considerations

- **Restricted File Types**: By default, it accepts only PDFs (`.pdf`) and HTML files. Additional validations (beyond MIME type checks) can be added for safer file handling.
- **Temporary File Storage**: The system saves a limited amount of state (pickled) and automatically cleans up older data after 24 hours.
- **CORS**: Currently open to all origins. In a production environment, this might be restricted to trusted domains.

8. Potential Extensions and Future Work

1. **Metadata-Aware Querying**: Incorporate structured metadata filters (e.g., by date or section).
2. **Advanced Summarization**: Implement advanced summarization or knowledge distillation for large documents.
3. **Automatic Document Refresh**: Automate re-fetching and re-indexing of an online resource if the source changes frequently.
4. **Security Hardening**: Implement authentication, request quotas, or secure token-based usage for the endpoints.

9. Conclusion

This RAG application combines modern NLP techniques (sentence embeddings, semantic search) with the power of an LLM (Google's "gemini-pro") to create an intelligent, user-friendly chat interface that can accurately answer questions using factual data from uploaded documents. With its modular design, transparent code organization, and extensible architecture, it provides a solid foundation for further enhancements in both model quality and system performance.