Technical Documentation

Multilingual PDF Retrieval-Augmented Generation (RAG) System

---

1. Introduction

This system is designed to process multilingual PDFs (Hindi, English, Bengali, Chinese) and answer queries based on their content. The solution integrates Optical Character Recognition (OCR), advanced vector-based search, and conversational AI, providing context-aware answers. It is scalable to large datasets (up to 1TB) and optimized for minimal latency.

---

2. System Architecture

High-Level Design

1. **Input Layer**:
    - Upload multilingual PDFs (both scanned and digital).
    - Extract text using OCR and standard text parsing.
2. **Processing Layer**:
    - **Text Extraction**: Extract raw text from PDFs using PyPDF2 and OCR.
    - **Text Chunking**: Break text into manageable chunks with overlap for context continuity.
    - **Vector Store**: Create embeddings using Google Generative AI and store them in FAISS for semantic retrieval.
3. **Query Engine**:
    - Perform keyword and semantic-based searches.

    - Use LangChain and Google Generative AI to provide detailed, context-aware responses.
4. **Output Layer**:
    - Display query results interactively in a Streamlit app.

System Components

1. **PDF Processing**:
    - Libraries: `PyPDF2`, `langchain.text_splitter`.
    - OCR for scanned PDFs and direct parsing for digital PDFs.
2. **Vector Database**:
    - **Technology**: FAISS (Facebook AI Similarity Search).
    - Stores document embeddings for fast and efficient similarity searches.
3. **Conversational AI**:
    - Framework: LangChain.
    - Model: Google Generative AI (Gemini-Pro).
4. **Front-End Interface**:
    - Framework: Streamlit.
    - Provides interactive PDF uploads and query functionality.

---

3. Technical Workflow

Step 1: PDF Upload

- Files are uploaded via Streamlit's sidebar.
- Supports multiple files.

Step 2: Text Processing

- Text extraction:
    - **Scanned PDFs**: OCR.
    - **Digital PDFs**: Direct text parsing.
- Text is chunked into blocks with overlap using `RecursiveCharacterTextSplitter` for context preservation.

Step 3: Vectorization

- Text chunks are converted to vector embeddings using `Google Generative AI Embeddings`.
- Vector store is created and managed using FAISS.

Step 4: Query Handling

- Queries are processed by LangChain's QA pipeline.
- Similarity search identifies relevant chunks from the vector store.
- Contextual responses are generated using Google Generative AI.

Step 5: Results Presentation

- Responses are displayed in the main app window.

---

4. Key Features

1. **Multilingual Support**:
    - Handles Hindi, English, Bengali, and Chinese PDFs.
2. **Hybrid Search**:
    - Combines semantic and keyword-based search techniques.

3. **Scalability**:
    - Processes large datasets and supports up to 1TB of data.
4. **Metadata Handling**:
    - Tracks processed documents and metadata for retrieval optimization.
5. **Minimal Latency**:
    - Optimized for small LLMs to reduce computational overhead.

---

5. Performance Evaluation

Metrics

1. **Query Relevance**:
    - Accuracy of results aligned with user queries.
2. **Latency**:
    - Measured response times; optimized for speed.
3. **Scalability**:
    - Tested on datasets of varying sizes.
4. **Fluency**:
    - Coherence and clarity of answers.

---

6. Setup and Deployment

1. **Installation**:
    - Install dependencies from `requirements.txt`.

bash
Copy code
```
pip install -r requirements.txt
```

2.
3. **Environment Variables**:
    - Set `GOOGLE_API_KEY` in an `.env` file.

Run Application:
bash
Copy code
```
streamlit run app.py
```

4.

---

7. Limitations and Future Work

Current Limitations:

- Limited OCR accuracy for poorly scanned documents.
- Dependency on Google Generative AI for embeddings.

Future Enhancements:

- Add support for more languages.
- Improve OCR accuracy for noisy datasets.
- Integrate caching for faster query responses.