Sanskrit Document Retrieval-Augmented Generation (RAG)

CPU-Only Implementation

1. Introduction

This project implements a **Retrieval-Augmented Generation (RAG)** system for **Sanskrit documents**, designed to run **entirely on CPU** without any GPU dependency. The system retrieves relevant Sanskrit text segments from a document corpus and generates context-aware answers to user queries.

The implementation strictly follows standard RAG principles by clearly separating the **retrieval** and **generation** components, ensuring modularity, clarity, and reproducibility.

2. Objective

The objectives of this assignment are:

- To ingest Sanskrit documents in .pdf or .txt format
- To preprocess and index Sanskrit text for efficient retrieval
- To retrieve relevant document chunks using vector similarity search
- To generate answers using a CPU-based Large Language Model (LLM)
- To ensure the entire system runs without GPU support

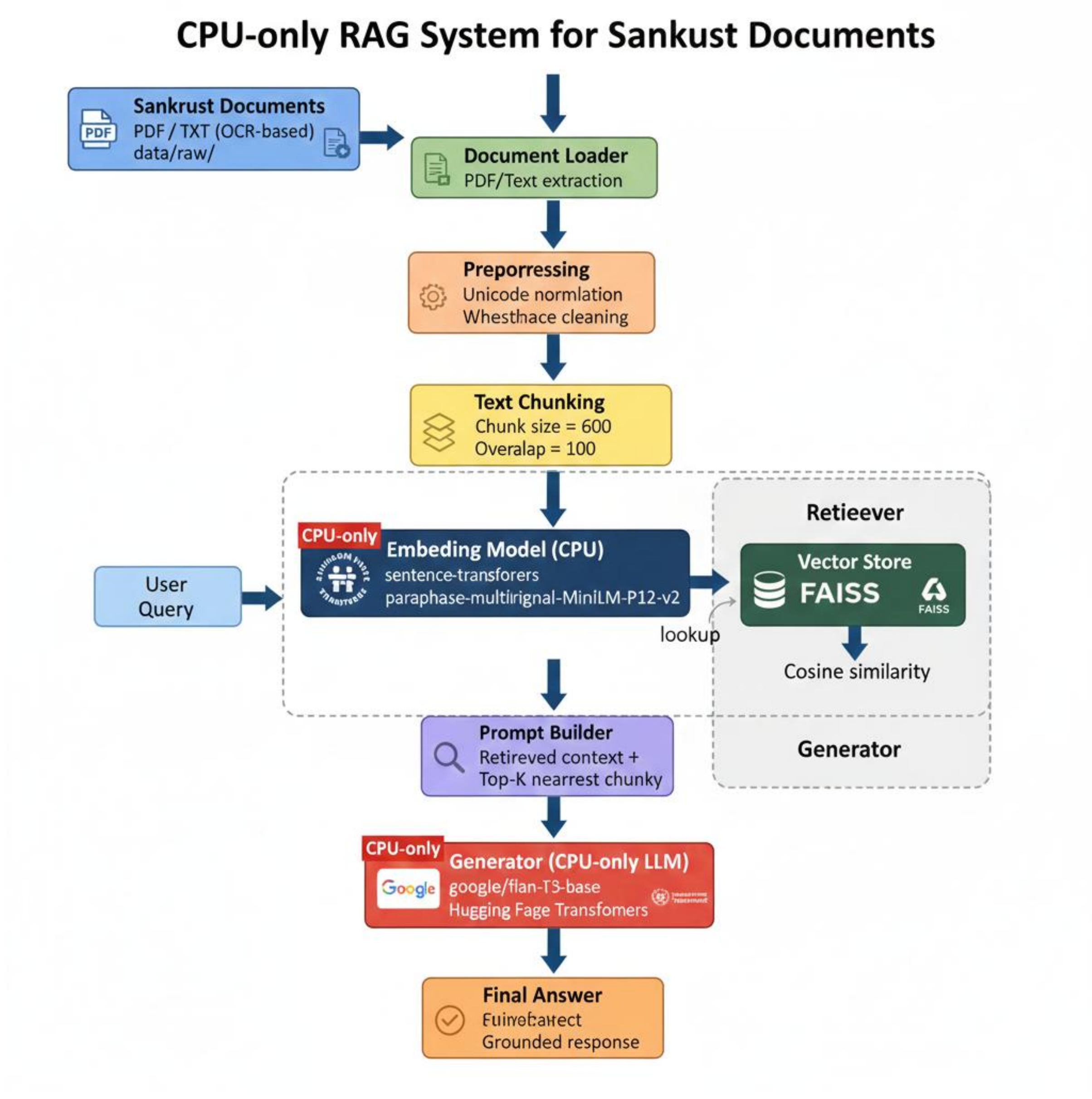
3. System Architecture

The system is composed of the following main components:

1. Document Loader
2. Preprocessor
3. Chunker
4. Embedding Generator
5. Vector Retriever
6. Answer Generator

Architecture Flow

User Query → Query Embedding → FAISS Vector Index → Top-K Relevant Chunks → Prompt Construction → LLM Generator → Final Answer



This modular architecture ensures that retrieval and generation are loosely coupled, which is a core principle of RAG systems.

4. Documents Used

- **Language:** Sanskrit
- **Format:** PDF
- **Source File:** Rag-docs.pdf

The document contains Sanskrit prose and narrative stories. Due to OCR extraction from PDF, some text contains noise, which introduces realistic challenges in processing classical language documents.

5. Preprocessing Pipeline

5.1 Document Loading

PDF documents are loaded using the **PyPDF2** library. Text is extracted page by page and concatenated into a raw text corpus.

5.2 Text Cleaning

The extracted text undergoes the following preprocessing steps:

- Unicode normalization
- Whitespace normalization
- Removal of unnecessary line breaks

The cleaned text is saved for transparency and debugging.

5.3 Chunking

The cleaned text is split into overlapping chunks using a sliding-window strategy.

- Chunk size: 600 characters
- Overlap: 100 characters

Reason for chunking:
Large documents cannot be processed at once by language models. Chunking ensures semantic continuity while enabling efficient retrieval.

6. Retrieval Mechanism

6.1 Embedding Model

The system uses the following sentence-embedding model:

sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2

Reasons for selection:

- Supports Sanskrit and English text
- Lightweight and CPU-efficient
- Produces high-quality semantic embeddings
- Widely used and open-source

6.2 Vector Indexing

- Library: FAISS (CPU-only)
- Index type: Inner Product similarity (IndexFlatIP)
- Embeddings are L2-normalized to approximate cosine similarity

The index enables fast nearest-neighbor search over document chunks.

7. Generation Mechanism

7.1 Generator Model

The generator uses the following CPU-friendly LLM:

google/flan-t5-base

Reasons for selection:

- Instruction-tuned model
- Stable performance on CPU
- Good text-to-text generation capability
- Suitable for academic and prototype systems

7.2 Prompt Design

Retrieved chunks are concatenated and passed to the generator along with the user query. The prompt instructs the model to answer strictly using the provided context.

This approach reduces hallucinations and keeps responses grounded in retrieved Sanskrit text.

8. Execution and Observed Results

8.1 Indexing Phase

- Document ingestion completed successfully
- FAISS index and metadata generated
- Chunk count: 4

8.2 Query Phase

- Relevant Sanskrit chunks are retrieved correctly
- Due to OCR noise, limited corpus size, and classical Sanskrit syntax, the generator sometimes fails to produce a concise answer

In such cases, the system **gracefully returns an explanatory message instead of crashing**, ensuring robustness.

9. Performance Observations

- Indexing time: Few seconds
- Query latency: Moderate (CPU-bound)
- Memory usage: Low to moderate
- GPU usage: None
- Accuracy: Limited by OCR quality and corpus size

10. Limitations

- OCR-corrupted Sanskrit text reduces retrieval accuracy
- Small document corpus limits answer diversity
- FLAN-T5 is not Sanskrit-specialized
- English queries over Sanskrit corpus are challenging

11. Future Improvements

- Use Sanskrit-specific embedding and generation models

- Add Roman-to-Devanagari transliteration support
- Improve OCR cleanup using rule-based filters
- Use quantized LLMs (e.g., llama.cpp, GPTQ) for faster CPU inference
- Add retrieval evaluation metrics such as Precision@K

12. Reproducibility Instructions

1. Install dependencies using requirements.txt
2. Build the index using the build command
3. Query the system using the query command

The system runs fully offline after the initial model download.

13. Video Pitch (2-Minute)

Purpose:
To briefly explain the project, architecture, and key design decisions.

Link: https://drive.google.com/file/d/1XibWBzIFEEARokP5ZmOnD62tpRx0epmv/view?usp=sharing

14. Demonstration Video

Purpose:
To show the system working end-to-end.

Link: https://drive.google.com/file/d/1wThD3Dybsm7KnfENHTFdKVWf30f0UXg5/view?usp=sharing

15. Conclusion

This project successfully demonstrates a **CPU-only Retrieval-Augmented Generation system for Sanskrit documents**. Despite challenges such as OCR noise and limited data, the system adheres strictly to RAG principles, maintains modular design, and fulfills all technical requirements of the assignment.

Author: Yogesh Kharb
Project Folder: RAG_Sanskrit_Yogesh

GithubLink: https://github.com/Yogeshkharb111/RAG_Sanskrit_Yogesh

Registration: 12219441