Technical Report

Sanskrit Document Retrieval-Augmented Generation (RAG) System (CPU Only)

Project Title:

Sanskrit Document Retrieval-Augmented Generation (RAG) System

Submitted By:

Swaroop Kumbhalwar

Folder Name:

`immverse_ai_rag_project`

---

1. Objective

The objective of this assignment is to design and implement a Retrieval-Augmented Generation (RAG) system capable of processing and answering queries based on Sanskrit documents. The system must operate fully on CPU-based inference without GPU usage.

---

2. System Description

This project implements an end-to-end RAG pipeline that performs the following operations:

1. Ingest Sanskrit documents (in `.txt` format)

2. Preprocess and chunk the documents

3. Index the chunks for efficient retrieval

4. Accept user queries in Sanskrit or transliterated text

5. Retrieve relevant context chunks from the indexed corpus

6. Generate coherent responses using a CPU-based Large Language Model (LLM)

The solution follows modular RAG principles with clear separation between the retriever and generator components.

---

3. System Architecture & Flow

3.1 High-Level RAG Flow

1. **Document Loader**

2. **Preprocessing + Chunking**

3. **Embedding Generation**

4. **Vector Index Creation (FAISS)**

5. **User Query Input (CLI)**

6. **Retriever → Top-k Context Chunks**

7. **Prompt Construction**

8. **LLM Generation**

9. **Final Answer + Sources Output**

3.2 Architecture Modules

A) Ingestion Module (`build_index.py`) – Offline Pipeline

- Loads Sanskrit `.txt` documents from `/data`

- Splits text into chunks

- Generates embeddings for chunks

- Creates FAISS vector index and saves it to `/code/faiss_index`

B) Query Module (`query.py`) – Online Pipeline

- Loads FAISS index from disk

- Retrieves most relevant chunks using vector similarity search + MMR

- Sends retrieved context and query to the generator LLM

- Prints answer with source evidence

---

4. Sanskrit Documents Used

The Sanskrit dataset consists of short stories/subhashitas stored in `.txt` format inside `/data/`.

Documents were separated story-wise for better retrieval quality.

Files used:

- `murkhabhritya.txt` – **मूर्खभृत्यस्य** (Shankhnad story)

- `devbhakta.txt` – **देवभक्तः कथा** (devotion + effort)

- `ghantakarna.txt` – **वृद्धायाः चार्तुयम् / घण्टाकर्णः कथा**

- `kalidasa.txt` – **चतुरस्य कालीदासस्य** (clever Kalidasa story)

- `sheetam.txt` – **शीतं बहु बाधति** (cold hurts very much)

Story-wise separation reduces context mixing and improves answer relevance.

---

5. Preprocessing Pipeline

5.1 Document Loading

Documents are loaded using:

- `DirectoryLoader`

- `TextLoader(encoding="utf-8")`

5.2 Chunking

Chunks are created using `RecursiveCharacterTextSplitter`.

Configuration:

- **chunk_size = 350**

- **chunk_overlap = 50**

Chunking improves retrieval accuracy and avoids exceeding model context/token limits.

5.3 Embedding Generation

Chunk embeddings are generated using:

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

Reason for selection:

- CPU-friendly

- multilingual semantic retrieval support

- effective for Sanskrit/Hindi-like scripts

---

6. Retriever Component

6.1 Vector Store

Vector database used:

- **FAISS (CPU)**

FAISS supports fast similarity search and efficient local storage.

6.2 Retrieval Strategy

Retriever uses:

- **MMR (Max Marginal Relevance)**

Benefits:

- retrieves diverse but relevant chunks
- reduces redundancy
- improves contextual grounding

---

7. Generator Component

7.1 LLM Model (CPU Only)

Generator model:

- `google/flan-t5-small`

Reason:

- lightweight model suitable for CPU inference

- faster than larger LLMs

- works well for QA-style generation

7.2 Prompting Strategy

A constrained prompt template is used to ensure:

- answers are generated only from retrieved context

- if answer is not present, system outputs:

  `सन्दर्भे उत्तरं न लभ्यते।`

The system also prints sources to improve transparency.

---

8. Performance Observations

8.1 Latency

- **FAISS retrieval latency:** fast (usually under 1 second)

- **LLM generation latency (CPU):** depends on CPU speed, typically a few seconds per query

8.2 Resource Usage

- Runs completely on CPU

- Resource usage mainly depends on:

  - embedding model memory

  - FAISS index size

  - FLAN-T5 model memory

Using `flan-t5-small` ensures reduced RAM usage and faster inference.

8.3 Relevance / Accuracy

Accuracy mainly depends on the retriever quality.

Improvements observed:

- separating documents story-wise reduces wrong retrieval

- MMR improves diverse context selection

Example:

- Query: `शंखनादः कः?`

- Correct retrieval from `murkhabhritya.txt`

- Generated response:

  शंखनादः गोवर्धनदासस्य भृत्यः (आज्ञापालकः) अस्ति।

---

9. Conclusion

The Sanskrit Document RAG System was successfully implemented as an end-to-end CPU-based pipeline. The system ingests Sanskrit documents, preprocesses and indexes them using embeddings + FAISS, retrieves relevant context, and generates answers using a CPU-based LLM. This implementation follows standard modular RAG architecture and satisfies the requirements of the assignment.

---

10. Future Improvements

- Add transliteration support (IAST / English input → Devanagari conversion)

- Add support for PDF ingestion (`PyPDFLoader`)

- Use Sanskrit-specific embedding model (if available) for higher retrieval accuracy

- Build a simple UI (Streamlit / Flask) for demonstration

- Add evaluation metrics (Top-k retrieval accuracy / BLEU-like score / response relevance scoring)