Sanskrit Retrieval-Augmented Generation (RAG) System – Technical Report

==

1. Abstract

This project presents a CPU-based Retrieval-Augmented Generation (RAG) system designed to answer queries from a Sanskrit text corpus. The system performs document preprocessing, text cleaning, chunking, embedding generation, vector indexing, and contextual generation using a local Large Language Model (LLM) through Ollama (phi3:mini). This report explains all components, methodology, architecture, and results.

2. Introduction

The goal of this internship assignment was to build a working RAG pipeline capable of answering questions from Sanskrit stories. Unlike standard LLMs, RAG ensures accurate information retrieval by combining vector search with generative models. This system runs entirely on CPU, fulfilling the assignment requirements.

3. Dataset Description

The dataset consists of Sanskrit prose passages extracted from:

- Rag-docs.docx (provided)

These include:

- ■■■■■■■■■■■■■■ ■■■

- ■■■■■■■■■■ ■■■■■■■■■■

- ■■■■■■■■■■■ ■■■

- Additional Sanskrit narratives

The corpus was converted from DOCX to plain text and used as the knowledge base.

4. Preprocessing

Preprocessing steps included:

1. DOCX → TXT conversion using python-docx

2. Removal of blank lines and inconsistent whitespace

3. UTF-8 normalization

4. Storing cleaned text into data/cleaned/

These steps ensured consistent formatting for downstream processing.

5. Text Chunking

To enable efficient vector search, the corpus was chunked into segments of approx. 300–350 characters. Each chunk was stored alongside metadata:

- chunk_id

- document_name

- original_text

This chunking is essential for retrieval accuracy.

6. Embeddings

We used the model:

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

This model supports Sanskrit and performs well on multilingual semantic similarity tasks.

Each chunk was converted into a dense vector and stored in:

- embeddings.npy (optional)

- faiss_index.bin (mandatory)

- metadata.json

7. Vector Index (FAISS)

FAISS (CPU version) was used to create a similarity search index:

- Index type: IndexFlatL2

- Number of vectors: Matches number of chunks

This allows fast nearest-neighbor search during query time.

8. Local LLM (Ollama – phi3:mini)

Instead of GPT4All (large downloads), the final system integrates:

Ollama model: phi3:mini

Advantages:

- Small (≈2.2 GB)

- CPU-friendly

- High-quality reasoning

- Runs offline

Ollama handles final natural language generation using retrieved chunks as context.

9. RAG Architecture

The architecture includes:

User Query → Embedding → FAISS Search → Top-k Chunks → LLM Prompt Construction → Final Answer

10. Implementation Summary

Main scripts:

- docx_to_txt.py : Converts DOCX to TXT

- clean_texts.py : Cleans text

- chunk_sanskrit.py : Splits text into chunks

- create_embeddings.py : Creates embeddings + FAISS index

- query_index.py : Tests retrieval

- rag_qa.py : Final pipeline (LLM + retrieval)

11. Results

Queries tested successfully:

- “■■■■■■■■■■■■■ ■■■■■■ ■■■■■■■ ■■■”

- “What is the moral of the bell story?”

Results were accurate and contextually grounded.

12. Limitations

- Limited dataset → only answers from provided stories

- Depends on embedding quality for retrieval

- CPU-only models are slower than GPU alternatives

13. Conclusion

This project successfully implements a complete RAG pipeline fulfilling all assignment requirements. It demonstrates practical skills in NLP preprocessing, embeddings, vector search, and local LLM integration.

14. References

- FAISS by Meta AI

- Sentence-Transformers library

- Ollama documentation
- LangChain (optional)