Sanskrit RAG System – Technical Report

1. System Architecture and Workflow

The Sanskrit Retrieval Augmented Generation RAG System is a fully local, CPU based application that allows users to query Sanskrit documents and receive responses in both Sanskrit and English. The system is designed for offline operation, ensuring complete data privacy by executing all components on the local machine.

Core Components

User Interface (Frontend)
The frontend is built using Streamlit in `app.py`. It provides an interactive chat style interface for submitting queries and viewing generated answers along with the retrieved source context.

RAG Engine (Backend)
The backend logic is implemented in `rag_engine.py` as a Python class named `SanskritRAG`. This component coordinates document retrieval and response generation using the LangChain framework.

Knowledge Base (Vector Store)
ChromaDB is used as the vector database to store embeddings of Sanskrit documents. It enables fast and efficient semantic similarity based retrieval.

Inference Engine (LLM)
LlamaCpp is used to run a quantized TinyLlama 1.1B model entirely on the CPU. This allows offline inference on low resource systems without GPU support.

Data Flow Overview

Document Ingestion Pipeline

1. Raw text or PDF documents are loaded.
2. Documents are divided into smaller semantic chunks.
3. Vector embeddings are generated for each chunk.
4. Embeddings are stored locally in ChromaDB.

Query Processing Flow

1. **User Input**
   The user submits a query in either Sanskrit or English through the Streamlit interface.
2. **Retrieval**
   The query is embedded, and the three most relevant document chunks are retrieved from ChromaDB using semantic similarity search.
3. **Augmentation**
   The retrieved chunks are combined with the user query using a predefined prompt template.

4. **Generation**
   The TinyLlama model generates an answer strictly grounded in the retrieved context. The response is produced first in Sanskrit, followed by an English translation.
5. **Output Display**
   The final answer and the corresponding source chunks are displayed to the user in the interface.

---

2. Sanskrit Document Corpus

The system is initialized with a curated collection of Sanskrit narratives and subhāṣitas that serve as the knowledge base.

Source File
`rag-docs.txt`

Content Categories

- **Parables**
  Short narrative texts such as *Mūrkha Bhṛtya* (The Foolish Servant) and *Vṛddhāyāḥ Cāturiyam* (The Old Woman's Cleverness).
- **Historical and Literary Anecdotes**
  Stories related to King Bhoja and the poet Kālidāsa, including *Śītaṁ Bahu Bādhate*.
- **Moral Narratives**
  Stories emphasizing karma, human effort, and wisdom, such as the account of a devotee waiting for divine help while rejecting human assistance.

Linguistic Structure

The corpus primarily consists of Sanskrit prose, with occasional English translations, annotations, and metadata such as glosses or transcription notes.

Dataset Size

- Approximately 20 KB of text
- Around 170 lines

---

3. Sanskrit Document Preprocessing

To ensure accurate retrieval and semantic integrity, a preprocessing pipeline optimized for Sanskrit text is employed.

Document Loader

Text files are loaded using `TextLoader` with UTF 8 encoding to preserve Devanagari characters.

Text Splitting Strategy

- **Tool**: RecursiveCharacterTextSplitter
- **Chunk Size**: 500 characters
- **Chunk Overlap**: 50 characters

Sanskrit Aware Separators

The splitter prioritizes separators in the following order:

- `||` for paragraph boundaries
- `|` for sentence boundaries
- Newline characters
- Spaces

This approach avoids breaking verses or sentences mid structure and maintains grammatical coherence.

Embedding Model

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

This multilingual and lightweight embedding model supports over fifty languages and performs well for Sanskrit, English, and mixed language semantic search tasks.

---

4. Retrieval and Generation Design

Retrieval Strategy

- **Method**: Semantic similarity search using cosine similarity
- **Top K Results**: 3
- **Vector Store**: ChromaDB persisted locally in the `./db` directory

Generation Strategy

- **Model**: `tinyllama-1.1b-chat-v1.0.Q4_K_M.gguf`
- **Quantization Level**: Q4_K_M

This configuration enables inference on systems with less than 1 GB of available RAM while maintaining acceptable coherence and contextual accuracy.

Prompt Structure

The system uses a structured prompt that injects retrieved context directly into the model input. The prompt instructs the model to rely solely on the provided context and produce answers in Sanskrit followed by English. If the answer is absent from the context, the model is instructed to explicitly state that it is not found.

Chain Configuration

A *Stuff* chain is used, where all retrieved chunks are directly concatenated and passed into the prompt context window.

---

5. Setup and Execution

Detailed installation and usage instructions are provided in `README.md`.

High level execution steps include:

1. Installing required dependencies.
2. Downloading the language model using `download_model.py`.
3. Running `ingest.py` to preprocess documents and build the vector database.
4. Launching the application using `app.py`.

---

6. Performance Evaluation

As the system operates entirely on CPU, performance varies based on available hardware. The architecture prioritizes efficiency, responsiveness, and ease of use.

Latency Metrics

- **Retrieval Time**: Under 0.5 seconds for small to medium datasets
- **Generation Time**: Approximately 2 to 10 seconds per query on a modern CPU

TinyLlama 1.1B offers significantly faster inference compared to larger models such as LLaMA 2 or Mistral 7B, enabling interactive usage without GPU acceleration.

Accuracy and Reliability

- High relevance in retrieval for keyword driven queries
- Effective handling of Devanagari Sanskrit text
- Strong adherence to retrieved context due to strict prompt constraints

Resource Consumption

- **RAM Usage**: Operates comfortably within 4 GB
- **Disk Usage**:
    - Model size approximately 670 MB
    - Vector database under 1 MB for the current dataset
    - Dependencies between 1 and 2 GB

Scalability Considerations

The current design is suitable for academic, prototype, and intern level projects and can handle several thousand documents locally. Scaling to significantly larger corpora would require deploying ChromaDB in server mode and upgrading the embedding and storage infrastructure.

Conclusion

The Sanskrit RAG System demonstrates that reliable, privacy focused, and multilingual question answering over classical Sanskrit texts is achievable on standard consumer hardware. By combining lightweight language models, Sanskrit aware preprocessing, and retrieval augmented generation, the system delivers accurate, context grounded responses without dependence on cloud based services.

Submitted By : Deep Bhagat