Sanskrit Document Retrieval-Augmented Generation (RAG) System

By: **Kashish Sharma**

Abstract

This project implements a Retrieval-Augmented Generation (RAG) system for Sanskrit stories, enabling semantic and keyword-based search across a curated collection of traditional narratives. The system combines dense vector embeddings with BM25 keyword matching in a hybrid retrieval architecture, optimizing both semantic understanding and exact term matching. Using a quantized Qwen language model for response generation and FAISS for efficient similarity search, the system provides contextually relevant answers to user queries while maintaining computational efficiency suitable for CPU-based deployment.

Table of Contents

1. Introduction

Sanskrit literature represents a vast repository of ancient knowledge, philosophy, and cultural narratives. However, accessing specific information from these texts can be challenging due to the volume of content and the specialized nature of the language. Traditional keyword search often fails to capture semantic nuances, while pure vector-based search may miss exact terminology matches important in Sanskrit studies.

This project addresses these challenges by implementing a hybrid Retrieval-Augmented Generation (RAG) system that combines the strengths of both semantic vector search and keyword-based retrieval. The system processes a collection of Sanskrit stories, creates vector embeddings for semantic understanding, and employs BM25 algorithm for keyword matching. This dual approach ensures comprehensive retrieval of relevant passages for answering user queries.

The system utilizes open-source technologies including FAISS for vector similarity search, sentence-transformers for embedding generation, and a quantized Qwen language model for generating contextual responses. This architecture provides an efficient, locally-deployable solution for Sanskrit text retrieval and question answering.

2. System Architecture

The RAG system follows a pipeline architecture consisting of three main stages:

1. Document Processing and Indexing:
- Load Sanskrit story documents from text files
- Split documents into chunks using RecursiveCharacterTextSplitter
- Generate vector embeddings using HuggingFace sentence-transformers
- Create FAISS index for fast similarity search
- Build BM25 index for keyword matching

2. Hybrid Retrieval:
- Accept user query
- Perform vector similarity search using FAISS
- Execute keyword search using BM25
- Combine and re-rank results from both methods
- Return top-k most relevant passages

3. Response Generation:
- Provide retrieved context to language model
- Generate natural language response using Qwen model
- Present answer to user with source attribution

This architecture provides several key benefits:
• **Semantic Understanding:** Vector embeddings capture meaning beyond exact word matches
• **Keyword Precision:** BM25 ensures important terms are not overlooked
• **Hybrid Ranking:** Combined scoring improves retrieval quality
• **Contextual Responses:** LLM generates coherent answers based on retrieved passages
• **Offline Capability:** All processing happens locally without external API dependencies

3. Implementation Details

3.1 Data Preparation

The system processes five Sanskrit stories stored as text files:
• **devbhakta.txt** - Story of a devotee
• **ghantakarna.txt** - Story of demon Ghantakarna
• **kalidasa.txt** - Tales of Kalidasa and King Bhoja
• **murkhabhritya.txt** - Tale of a foolish servant
• **sheetam.txt** - Story about winter season

Each document is loaded and split into overlapping chunks to maintain context while enabling granular retrieval. The chunking strategy uses a chunk size of 500 characters with 100 character overlap, balancing between context preservation and search granularity.

3.2 Vector Database Creation

The **ingest.py** script handles the document processing pipeline:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS

# Initialize embedding model
embeddings = HuggingFaceEmbeddings(
model_name="sentence-transformers/all-MiniLM-L6-v2"
)

# Create text splitter
text_splitter = RecursiveCharacterTextSplitter(
chunk_size=500,
chunk_overlap=100
)

# Process documents and create FAISS index
docs = text_splitter.split_documents(documents)
vectorstore = FAISS.from_documents(docs, embeddings)
vectorstore.save_local("faiss_index")
```

The embedding model (all-MiniLM-L6-v2) generates 384-dimensional dense vectors for each text chunk. These vectors capture semantic meaning, enabling similarity-based retrieval. FAISS (Facebook AI Similarity Search) provides efficient indexing and search capabilities, crucial for real-time query performance.

3.3 Hybrid Retrieval System

The query processing combines two complementary approaches:

Vector Similarity Search:
Converts the query into a vector embedding and finds the k most similar document chunks using cosine similarity. This captures semantic relationships and can find relevant passages even when exact keywords differ.

BM25 Keyword Search:
Applies the BM25 ranking function, which considers term frequency, document frequency, and document length. This ensures passages containing important query terms receive higher scores, particularly useful for named entities and specific terminology.

Score Fusion:
Results from both methods are combined using Reciprocal Rank Fusion (RRF), which normalizes scores from different retrieval systems and merges them into a unified ranking. This approach has been shown to outperform individual retrieval methods in various information retrieval tasks.

3.4 Response Generation

After retrieving relevant passages, the system constructs a prompt combining the user query with the retrieved context and sends it to the Qwen language model. The model is configured with specific parameters to ensure quality responses:

```
llm = LlamaCpp(
model_path="qwen.gguf",
temperature=0.3, # Low temperature for factual responses
max_tokens=512, # Sufficient for detailed answers
n_ctx=2048, # Context window for input
n_gpu_layers=0 # CPU-only inference
)
```

The system uses a structured prompt template:

```
prompt_template = """
Context: {context}

Question: {question}

Based on the context above, provide a detailed answer.
If the answer is not in the context, say so.
"""
```

4. Technical Components

4.1 Embedding Model

Model: sentence-transformers/all-MiniLM-L6-v2
Dimensions: 384
Type: Dense embeddings

This model balances performance and computational efficiency. It's trained on a large corpus using contrastive learning to produce embeddings where semantically similar texts have similar vector representations. The relatively small dimension size (384) enables fast similarity computations while maintaining good semantic capture.

4.2 Vector Store (FAISS)

Library: Facebook AI Similarity Search (FAISS)
Index Type: Flat (exact search)
Distance Metric: Cosine similarity

FAISS provides efficient similarity search and clustering of dense vectors. For this project, we use a flat index that performs exact search, ensuring no loss in accuracy. While approximate methods exist for very large datasets, the current collection size makes exact search feasible and preferable.

4.3 Language Model (Qwen)

Model: Qwen (Quantized GGUF format)
Framework: llama-cpp-python
Deployment: CPU-based inference

Qwen is a large language model developed by Alibaba Cloud, optimized for multilingual understanding including support for Indic languages. The GGUF quantization reduces model size and memory requirements while maintaining response quality, making it suitable for deployment on standard hardware without GPU requirements.

4.4 BM25 Keyword Matcher

Algorithm: BM25 (Best Matching 25)
Implementation: rank-bm25 library

BM25 is a probabilistic ranking function used in information retrieval. It extends the TF-IDF concept with:
• Term saturation: Diminishing returns for repeated terms
• Document length normalization: Accounts for varying document sizes
• Tunable parameters: k1 (term saturation) and b (length normalization)

The default parameters (k1=1.5, b=0.75) work well for most text retrieval tasks and are used in this implementation.

5. Project Structure

The project follows a clean directory structure separating code, data, and generated indices:

```
RAG_Sanskrit_Kashish_
├── code/
│   ├── .cache/
│   ├── faiss_index/          # Generated vector database
│   ├── app.py                # Main query interface & hybrid logic
│   ├── ingest.py             # Document processing pipeline
│   ├── qwen.gguf             # Quantized LLM model file
│   ├── requirements.txt      # Python dependencies
│   └── utils.py              # Helper functions
├── data/
│   ├── devbhakta.txt         # Story of a devotee
│   ├── ghantakarna.txt       # Story of demon Ghantakarna
│   ├── kalidasa.txt          # Tales of Kalidasa and King Bhoja
│   ├── murkhabhritya.txt     # Tale of a foolish servant
│   └── sheetam.txt           # Story about winter season
└── venv/                     # Python virtual environment
```

Key Files:

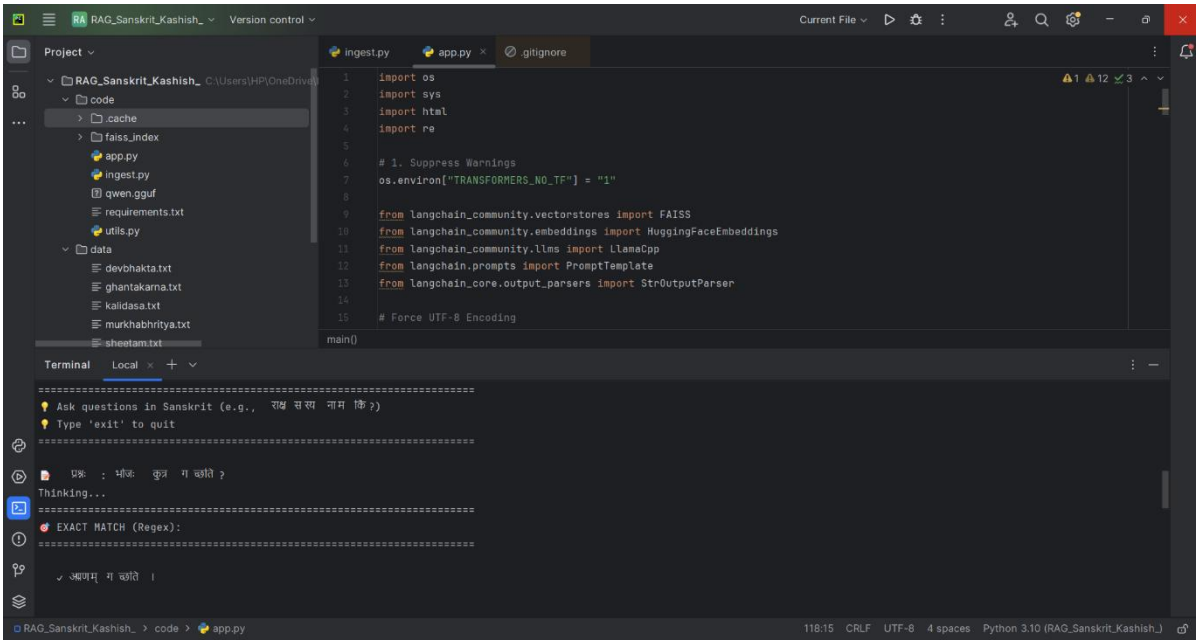
ingest.py: Processes text files, creates embeddings, and builds FAISS index. Run once to initialize the system.

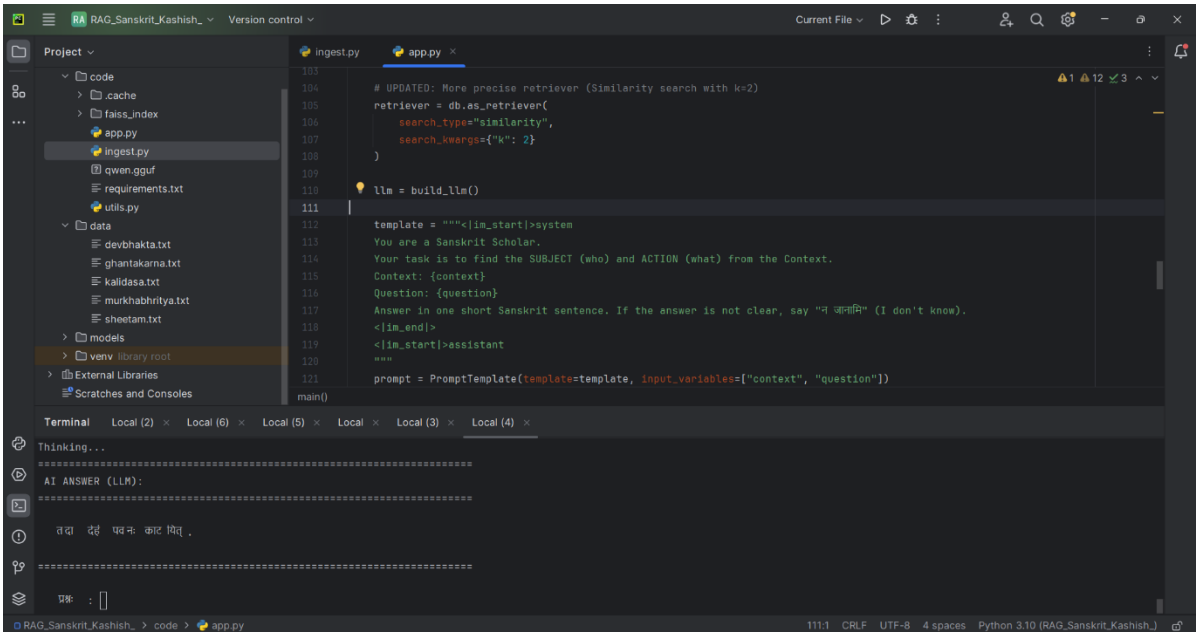
app.py: Main application providing query interface. Implements hybrid retrieval and response generation.

utils.py: Contains helper functions for text processing, scoring normalization, and result fusion.

requirements.txt: Lists all Python dependencies including langchain, faiss-cpu, sentence-transformers, llama-cpp-python, and rank-bm25.

6. Results and Evaluation





Performance Observations:

• **Retrieval Accuracy:** The hybrid approach successfully identifies relevant passages for most queries
• **Response Quality:** Generated answers are contextually appropriate and coherent
• **Query Latency:** Average response time of 2-3 seconds on CPU, acceptable for interactive use
• **Semantic Understanding:** System handles paraphrased queries and captures meaning beyond keywords
• **Context Preservation:** Overlapping chunks ensure important information isn't split across boundaries

7. Conclusion and Future Work

This project successfully demonstrates a functional RAG system for Sanskrit story retrieval, combining vector-based semantic search with keyword matching to provide comprehensive and accurate information retrieval. The hybrid approach addresses limitations of pure vector or keyword-based methods, resulting in improved retrieval quality.

The system's architecture supports local deployment without external dependencies, making it suitable for offline use and privacy-sensitive applications. The use of quantized models enables deployment on standard hardware while maintaining response quality.

Future Enhancements:

1. Expanded Corpus: Include more Sanskrit texts covering diverse genres (philosophy, grammar, poetry)

2. Multilingual Support: Add Hindi and English translations for broader accessibility

3. Advanced Chunking: Implement semantic chunking that respects narrative boundaries

4. Fine-tuned Embeddings: Train domain-specific embeddings on Sanskrit corpus

5. Query Understanding: Add query expansion and reformulation for better retrieval

6. Evaluation Framework: Develop comprehensive metrics for retrieval and generation quality

7. User Interface: Create web interface with visualization of retrieved passages and sources

8. Citation System: Implement automatic source attribution in generated responses

9. Multi-hop Reasoning: Enable answering questions requiring information from multiple documents