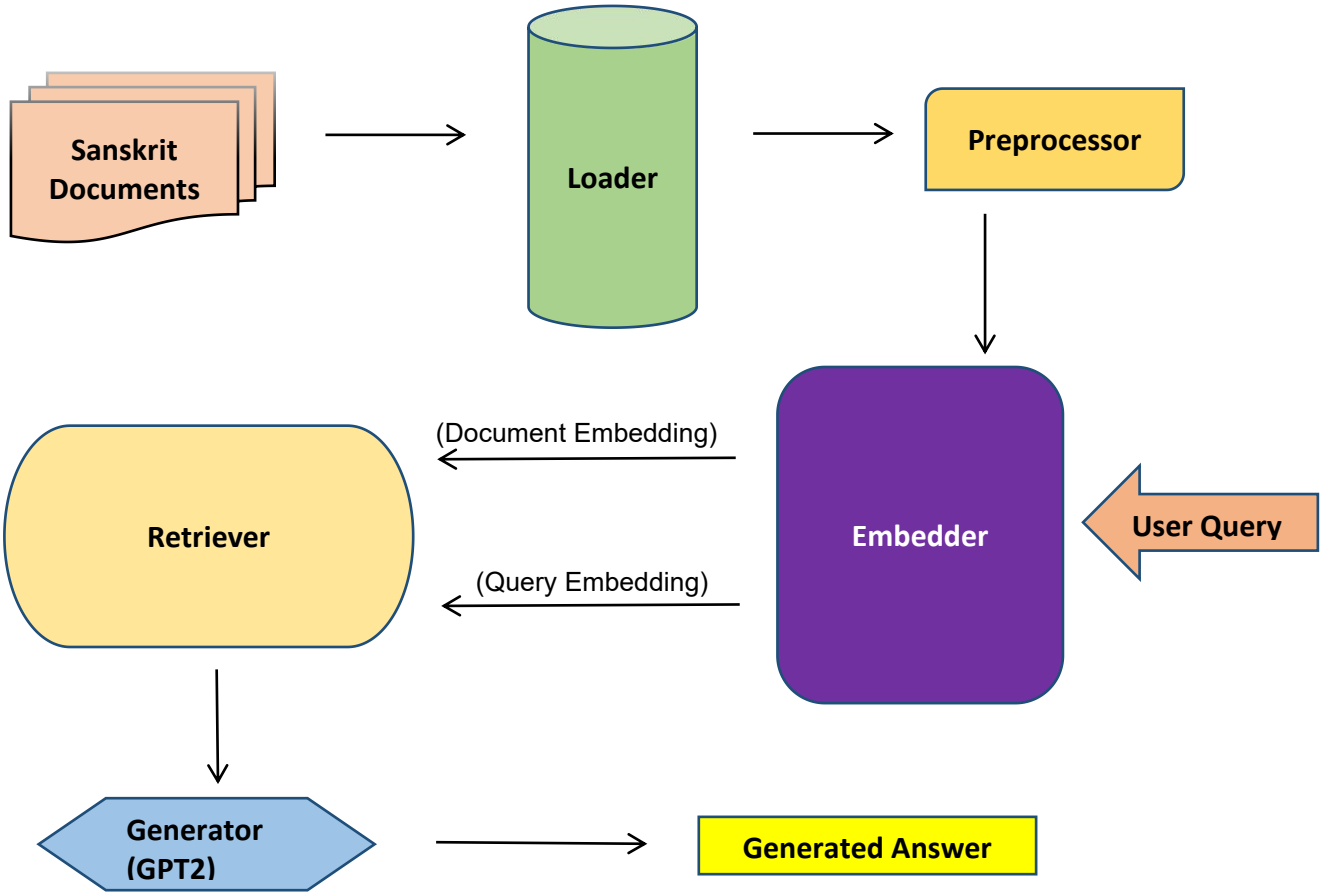
Sanskrit Document Retrieval-Augmented Generation (RAG) System

Reported by: GANGESHWAR

Introduction:

Sanskrit, one of the oldest Indo-European languages, is rich in grammar, morphology and classical literature. However, the scarcity of large annotated data-sets makes Sanskrit NLP particularly challenging. The project integrates classical NLP steps with modern machine learning models, enabling a hybrid approach that retrieves relevant textual evidence and uses a transformer-based model (GPT-2) to generate context- aware answers. To overcome data scarcity, Retrieval-Augmented Generation (RAG) combines information retrieval with language generation, enabling accurate responses even in low-resource settings. The entire system is designed using Python and popular NLP libraries such as SentenceTransformers and scikit-learn.

System Architecture:



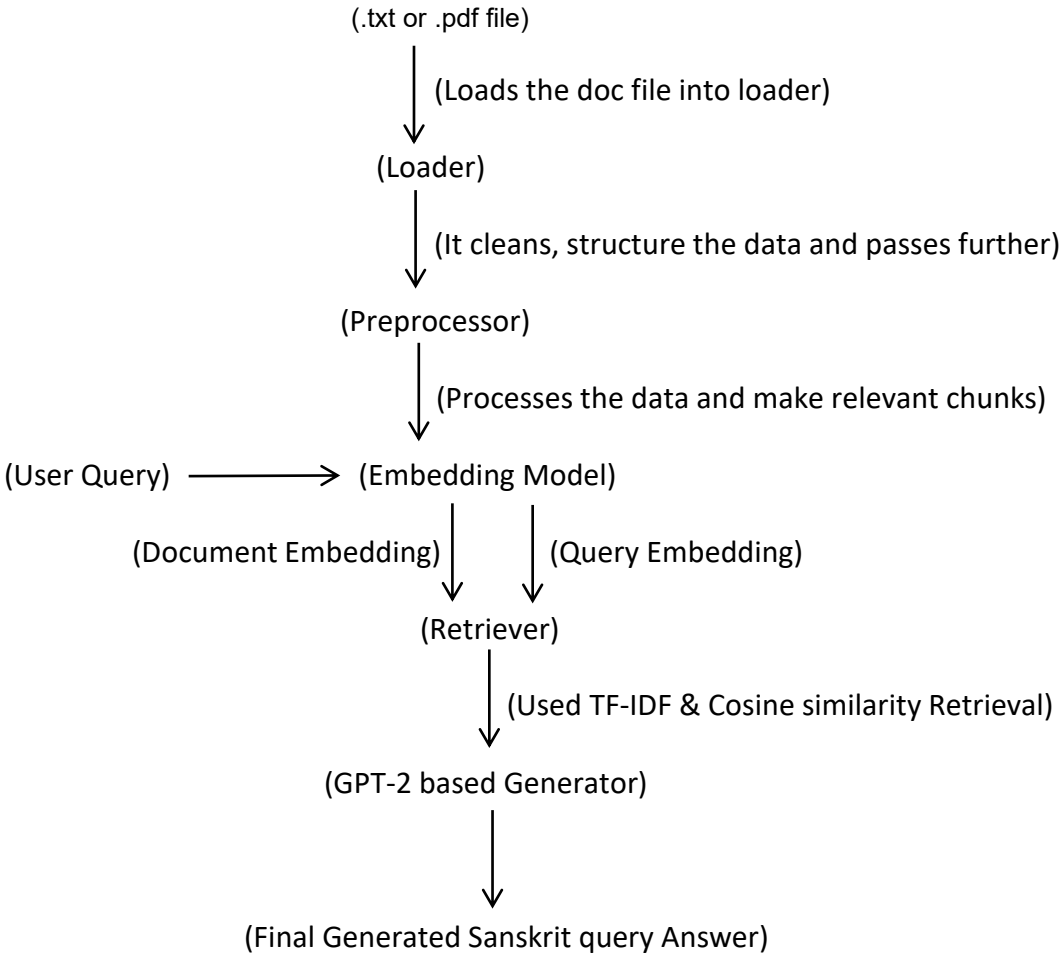
System consists of 5 main components which are as follows;

- **Loader:** It ingest the document and then convert that raw data from various sources into a format that AI models can process.
- **Preprocessor:** It cleans the ingested data, structure it and split it into chunks which are easy to manage. This makes our data RAG-ready by removing irrelevant information, standardizing the formats and creating meaningful chunks that the system can more effectively retrieve and use to generate more accurate and relevant responses.
- **Embedder:** It converts the text into numerical vectors also called as Vector embeddings which catches the semantic meaning and tells what is the exact meaning of the sentence.
- **Retriever:** It acts as an intelligent search engine that finds the most relevant and up to date information from an external knowledge base.

- **Generator:** It is the large language model (LLM) that produces a final, coherent response based on a user's query and the relevant context retrieved by the system's retriever component.

Flowchart of the Pipeline:

The following flowchart illustrates the logical sequence of operations performed by the Sanskrit RAG system—from query input to final answer generation.



Retrieval and Generation Mechanism:

- **Retrieval Mechanism:** The retrieval phase consists of few processes i.e.,

The **loader.py** file gathers all the sanskrit document inside the …data/sample_sanskrit_docs/ It accepts .txt, .pdf or .docx files and extract the clean text and ensure cross-platform path resolution using 'pathlib'. This produces a long string of document.

Then the **preprocessor.py** file performs the text sanitization to prepare the document for NLP processing:

#Removal of Non-Devnagari characters

text = re.sub(r"[^\u0900-\u097F\w\s]", " ", text)

This ensures only valid Sanskrit text and removes noise symbols, punctuations and special characters.

#WhiteSpace removal

text = re.sub(r"\s+", " ", text)

Removes all the whitespaces.

After this preprocessor creates the chunks of text for semantic retrieval.

Now code inside **embedder.py** file loads "SentenceTransformer Mini LM-L6-v2" i.e.,

model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

This creates vector embeddings for each chunk Although you generate dense embeddings, the retrieval step you actually use is TF-IDF retrieval, not dense embedding similarity.Dense embeddings are loaded but not used in the ranking.

Now in **retriever.py,** the TF-IDF and cosine similarity function encodes the rarity across the corpus and term importance in chunk. When the user entered the query the same TF-IDF vocabulary is applied and program calculate the Cosine Similarity. Higher the cosine similarity value, the query and chunk share similar important terms.

The retriever finds the relevant information and data from the provided document on the basis of TF-IDF and Cosine similarity and produces the best output for Generator.

- **Generation Mechanism:** After retrieval, the system moves into text generation using GPT-2. Then the relevant Sanskrit text chunks are selected by the retriever, they are combined with the user's query to form a structured prompt. This prompt typically includes two components: the retrieved context and the explicit question. The integration of context ensures that the language model does not rely purely on prior training but instead generates text grounded in actual documents. In this project, the GPT-2 transformer model is used as the generator. GPT-2 is an autoregressive language model, meaning it predicts the next token in a sequence based on all previously generated tokens. When provided with a prompt containing Sanskrit passages, GPT-2 leverages these examples to infer writing patterns, vocabulary, and semantic relationships. Although GPT-2 is primarily trained on English data, it adapts well when contextualized, allowing it to produce meaningful Sanskrit responses. The generation process continues until the model reaches a token limit or detects a natural stopping point. The final output is then decoded from token IDs into readable Devanagari text. This combination of retrieval-grounded information and generative reasoning enables the system to produce coherent, relevant answers.

Performance Observations:

The overall performance of the RAG system is efficient and responsive for typical Sanskrit question-answering tasks. Most of the processing time is spent in the generation stage, as GPT-2 requires more computation while producing the final answer. Retrieval using TF-IDF is extremely fast and consistently identifies relevant text chunks because Sanskrit vocabulary tends to have clear lexical roots and limited ambiguity. This ensures that the model receives appropriate context for generating meaningful output.

In terms of accuracy, the system produces coherent and contextually aligned Sanskrit responses when the retrieved chunks are relevant. Retrieval accuracy is generally high, especially when the top 3 matching chunks are considered. The GPT-2 model, although not trained specifically on Sanskrit, performs surprisingly well when guided by retrieved text, resulting in fewer hallucinations and more grounded answers.

Resource usage remains moderate. The models require around 1 GB of RAM and operate smoothly on CPU-only environments. Preprocessing and retrieval have negligible computational cost, while the generation step is the primary factor influencing latency. Overall, the system balances accuracy, speed, and resource consumption effectively for a classical-language RAG application.

Conclusion:

This project demonstrates that Retrieval- Augmented Generation can be effectively applied to low- resource languages like Sanskrit. The hybrid approach balances retrieval precision with generative flexibility. The architecture is modular, extensible, and capable of integrating more advanced models in the future such as LLaMA- 3, Mistral, or encoder-decoder architectures.