Technical Report on Website-Based Retrieval-Augmented Generation System

Introduction

This project implements a complete Retrieval-Augmented Generation (RAG) pipeline that transforms unstructured website content into a structured knowledge base capable of answering natural language questions. The motivation behind this work is to bridge the gap between static website information and interactive AI-powered search. By combining web crawling, natural language processing, embeddings, and large language models, the system enables context-grounded answers backed by verifiable sources.

📦 Install Dependencies

```
# --- Install required libraries: LangChain, ChromaDB, HuggingFace, BeautifulSoup ---
!pip -q install -U langchain-community langchain-text-splitters chromadb sentence-transformers beautifulsoup4
```

📥 Import Libraries and Set Parameters

```
# --- Import necessary modules and set base URL + crawl depth ---
from langchain_community.document_loaders import RecursiveUrlLoader
from bs4 import BeautifulSoup
from pathlib import Path
import re
```

🌐 Define Base URL and Depth

Website Crawling and Data Collection

The first stage of the system focuses on data ingestion from a target website. A recursive crawler is configured to start from a base URL and follow links within the same domain to a specified depth. This ensures that the crawl covers the homepage as well as its linked subpages, but avoids wandering into external sites or irrelevant sections. During this process, the crawler filters out unwanted resources such as PDFs, images, or compressed files. This selective crawling guarantees that the collected data consists of textual information that can later be processed for semantic search.

```
# --- Starting point of the crawl and how deep to follow links ---
START_URL = "https://www.educosys.com"
MAX_DEPTH = 2
```

🧹 HTML Cleaning Function

HTML Cleaning and Text Extraction

The raw content gathered from the crawler contains a significant amount of noise in the form of scripts, navigation bars, headers, footers, and style elements. To address this, the project integrates an HTML parser that removes these non-essential elements and extracts only the visible, human-readable text. The parser also normalizes the content by eliminating empty lines and redundant whitespace. Additional filters are applied to discard very short documents that do not carry enough information. At the end of this stage, the system has a set of clean, readable texts that accurately represent the website's knowledge base.

```
# --- Custom extractor using BeautifulSoup to remove scripts, headers, footers ---
def bs4_extractor(html: str) -> str:
    ...
```

🔎 Recursive Web Crawler Setup

```
# --- Configure recursive loader to crawl only inside domain and skip binary files ---
loader = RecursiveUrlLoader(
    url=START_URL,
    max_depth=MAX_DEPTH,
    extractor=bs4_extractor,
    ...
)
```

📑 Load and Filter Documents

```
# --- Load crawled pages and filter out irrelevant/short documents ---
docs_raw = loader.load()
docs = [...]
```

🎀 Document Chunking

Document Splitting

Large web pages often exceed the input size limits of modern language models, making it necessary to divide them into smaller, manageable pieces. The project achieves this by implementing a recursive text splitting strategy. Each document is segmented into chunks of approximately 1,200 characters, with an overlap of 150 characters to maintain continuity of meaning across boundaries. The splitter prioritizes natural divisions such as paragraphs and line breaks before resorting to character-level segmentation. By doing so, the system ensures that each chunk is semantically coherent while still being within the acceptable length for embeddings and retrieval.

```
# --- Split documents into manageable chunks with overlap ---
from langchain_text_splitters import RecursiveCharacterTextSplitter
splitter = RecursiveCharacterTextSplitter(...)
chunks = splitter.split_documents(docs)
```

👀 Inspect Chunks

```
# --- Print sample chunks to verify splitting ---
print(chunks[0])
print(chunks[1])
print(chunks[2])
```

🧠 Generate Embeddings

Embeddings and Semantic Representation

Once documents are split, the next task is to convert them into mathematical representations that capture semantic meaning. This is achieved using a sentence-transformer model provided by HuggingFace. The chosen model, all-MiniLM-L6-v2, is a lightweight but high-performing embedding generator that encodes each chunk into a dense vector. These embeddings allow the system to compare pieces of text based on their meaning rather than exact word matches. Computation can be accelerated using GPU resources when available, which is particularly useful in environments such as Google Colab.

```
# --- Initialize HuggingFace embedding model (GPU/CPU) ---
from langchain_community.embeddings import HuggingFaceEmbeddings
emb = HuggingFaceEmbeddings(...)
```

🗂️ Store in ChromaDB

Vector Storage in ChromaDB

The generated embeddings, along with their associated metadata, are stored in a Chroma vector database. Chroma is optimized for similarity search and enables fast retrieval of semantically related text chunks. Each entry in the database includes both the vector representation and the metadata such as the source URL. This ensures that any answer generated later can be directly linked back to the original website page. Persistence is enabled so that the database can be reused without having to regenerate embeddings each time.

```
# --- Create vector store from chunks and persist locally ---
from langchain_community.vectorstores import Chroma
vectorstore = Chroma.from_documents(...)
retriever = vectorstore.as_retriever(search_kwargs={"k": 5})
```

🤖 RAG Answer Function

Retrieval Mechanism

To answer user queries, the project leverages a retriever built on top of the Chroma database. When a question is posed, the retriever searches for the top five most semantically relevant chunks by comparing the query embedding with those stored in the database. This retrieval step narrows down the knowledge space and ensures that the language model only receives highly relevant context. By filtering content in this way, the system avoids irrelevant noise and improves both the accuracy and efficiency of the response.

```
# --- Define RAG function: retrieve context, build prompt, generate answer
---
from google.colab import ai
def rag_answer(question, k=5, max_chars=1200):
    ...
```

🧪 Test the System

RAG Answer Generation

The core functionality of the project lies in the RAG answering function. Once relevant chunks are retrieved, they are concatenated into a structured context block alongside the user's question. A carefully designed prompt instructs the language model to respond exclusively using the provided context, reducing the risk of hallucination. The project utilizes Google Colab's built-in AI service to generate responses, which eliminates the need for additional API keys. The output consists of two parts: the generated answer and a list of source URLs. This dual output ensures not only informative responses but also transparency, since users can verify the information against the original website.

```
# --- Ask a sample question and print answer with sources ---
answer, sources = rag_answer("Is neural networks part of the courses
offered?")
print(answer)
print("\nSources:")
for s in sources: print("-", s.metadata.get("source",""))
```