RAG PDF Chatbot

Overview:

A lightweight Retrieval-Augmented Generation (RAG) chatbot that allows users to upload a PDF, process it using chunked vector embeddings (FAISS), and query the document using natural language. This project uses:

- Flan-T5 Small as the language model
- FAISS for efficient document vector similarity search
- LangChain for orchestrating retrieval and generation
- Gradio for the web interface
- Hugging Face Spaces for public deployment

🚀 **Features:**

- Upload any PDF file and get insights instantly
- Uses vector embeddings for semantic search
- Fast and efficient QA using Flan-T5
- Clean, modern Gradio UI
- Ready for deployment on Hugging Face Spaces

Public Deployment (Hugging Face Spaces):

This app is deployed on Hugging Face Spaces. Click here to try it live (HS).

Github: https://github.com/kamalesh003/RAG-PDF-Chatbot

To deploy it yourself:

1. Create a new Space **(type: Gradio-free 16GB CPU RAM)**
2. Upload all project files
3. Include **requirements.txt , utils.py , config.py** and **app.py.**
4. The Space will automatically launch.

Tech Stack:

- transformers
- langchain
- faiss-cpu
- PyMuPDF
- gradio
- sentence-transformers

MODULE - 1: LangChain Integration (config.py):

This code snippet demonstrates how to load and configure a **sequence-to-sequence (seq2seq)** language model **(google/flan-t5-small)** using HuggingFace transformers and set up a text generation pipeline. It also shows how to load a sentence embedding model **(sentence-transformers/all-MiniLM-L6-v2)** and prepare a prompt template using LangChain for question answering based on document context.

Configuration Parameters

- **`LLM_MODEL_NAME` (str):** The HuggingFace model ID for the language model. Here it uses the Flan-T5-small model optimized for seq2seq tasks.

- **`EMBEDDING_MODEL_NAME` (str):** The sentence-transformer model ID used for producing vector embeddings from text.

- **`CHUNK_SIZE` (int):** Intended size of text chunks for splitting long documents (not directly used in this snippet).

- **`CHUNK_OVERLAP` (int):** Number of tokens overlapping between chunks to maintain context coherence (not used here directly).

- **`MAX_INPUT_TOKENS` (int):** Maximum input tokens allowed (not used directly here).

- **`MAX_NEW_TOKENS` (int):** Maximum number of tokens the model can generate in response.

- **`TOP_K` (int):** Parameter to restrict the number of tokens considered for generation (not used here explicitly).

Steps and Components:

i.) Load Tokenizer and Model

```
tokenizer = AutoTokenizer.from_pretrained(LLM_MODEL_NAME)
model = AutoModelForSeq2SeqLM.from_pretrained(LLM_MODEL_NAME)
```

Purpose: Load the pretrained tokenizer and seq2seq model from HuggingFace's model hub using the specified model name.The tokenizer converts input text into token IDs compatible with the model.The model is used to generate text outputs conditioned on inputs.

ii.) Create a Text Generation Pipeline

```
gen_pipeline = pipeline(
        "text2text-generation",
        model=model,
        tokenizer=tokenizer,
        max_new_tokens=MAX_NEW_TOKENS,
        truncation=True,)
```

Purpose: Create a HuggingFace pipeline for the task of text-to-text generation.The pipeline abstracts the process of tokenizing input, running the model, and decoding output. **max_new_tokens**, limits the maximum number of tokens generated in response.**truncation=True**, ensures inputs longer than the model max length are truncated.

iii.) Wrap the Pipeline for LangChain :

```
llm = HuggingFacePipeline(pipeline=gen_pipeline)
```

Purpose:Integrate the HuggingFace pipeline into LangChain's HuggingFacePipeline wrapper.This wrapper allows LangChain to interact with the HuggingFace pipeline as a Language Model (LLM) compatible with LangChain workflows.

iv.) Load Sentence Embedding Model

```
embedding_model=HuggingFaceEmbeddings(model_name=EMBEDDING_MODEL_NAME)
```

Purpose: Load a transformer-based sentence embedding model via LangChain's HuggingFaceEmbeddings wrapper.This model encodes text into fixed-size vector embeddings useful for semantic search, similarity, or retrieval tasks.

v.) Define a Prompt Template for Question Answering

```
prompt_template = """Use the following extracted parts of a document to answer the question.
Provide a concise, precise, and accurate answer. If you don't know, say "I don't know".
Context:
{context}
```

```
Question:
{question}
Answer:"""
```

Purpose: This is a prompt that guides the LLM on how to generate the answer based on extracted document context.It specifies the format: it expects a **`context`** and a **`question`** input.If the model does not have enough information, it should respond with "**`I don't know`**".

vi.) Create LangChain PromptTemplate Object:

```
PROMPT = PromptTemplate(
        template=prompt_template,
        input_variables=["context", "question"],
)
```

Purpose: Converts the raw string template into a LangChain PromptTemplate object.This allows the prompt to be dynamically filled with different contexts and questions during inference.

MODULE - 2: PDF Question Answering with LangChain, HuggingFace, and FAISS (utils.py):

This code snippet implements a PDF document question-answering system. It allows users to upload a PDF, process and split its contents, create a vector search index using FAISS embeddings, and then run a RetrievalQA chain to answer questions based on the document content.

Key Components

Imports & Configuration

```
import tempfile
import os
from langchain.chains import RetrievalQAWithSourcesChain
from langchain_community.vectorstores import FAISS
from langchain_community.document_loaders import PyMuPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

from config import (
    tokenizer,
    llm,
    embedding_model,
    CHUNK_SIZE,
    CHUNK_OVERLAP,
    TOP_K,
    MAX_INPUT_TOKENS,
    PROMPT,
)
```

- **`tempfile` and `os`:** For safely handling temporary PDF file storage.
- **`RetrievalQAWithSourcesChain`:** LangChain chain for QA with source referencing.
- **`FAISS`:** A fast vector store for semantic similarity search.
- **`PyMuPDFLoader`:** Loads PDF pages as documents.
- **`RecursiveCharacterTextSplitter`:** Splits large texts into manageable chunks preserving context.
- **`config`:** External module that provides pretrained tokenizer, LLM, embedding model, and parameters.

Global Variable:

```
qa_chain = None
```

Stores the initialized QA chain to be used for question answering after PDF processing.

Functions

i.) `truncate_context:`

```
def truncate_context(text, max_tokens=MAX_INPUT_TOKENS):
    tokens = tokenizer.encode(text, truncation=True, max_length=max_tokens)
    return tokenizer.decode(tokens, skip_special_tokens=True)
```

Purpose: Ensures generated text answers do not exceed a specified token length.

Input:

- **`text` (str):** Raw answer text.
- **`max_tokens` (int):** Maximum allowed tokens (default from config)

Output: Truncated and decoded string respecting token limits.

ii.) `create_qa_chain:`

```
def create_qa_chain(llm, retriever):
    return RetrievalQAWithSourcesChain.from_chain_type(
        llm=llm,
        retriever=retriever,
        chain_type="stuff",
        chain_type_kwargs={
```

```
        "prompt": PROMPT,
        "document_variable_name": "context"
    },
)
```

Purpose: Creates a LangChain retrieval-based QA chain that uses the LLM and a retriever to answer questions.

Parameters:

- **`llm`:** The language model pipeline wrapped for LangChain.
- **`retriever`:** A retriever object that fetches relevant documents based on query similarity.

Returns: A ready-to-use **`RetrievalQAWithSourcesChain`** object configured with a custom prompt.

iii. `Process_pdf:`

```
def process_pdf(pdf_file):
    global qa_chain
    with tempfile.NamedTemporaryFile(delete=False, suffix=".pdf") as tmp:
        tmp.write(pdf_file)
        tmp_path = tmp.name

    loader = PyMuPDFLoader(tmp_path)
    docs = loader.load()

            splitter = RecursiveCharacterTextSplitter(chunk_size=CHUNK_SIZE, chunk_overlap=CHUNK_OVERLAP)
    split_docs = splitter.split_documents(docs)

    vectordb = FAISS.from_documents(split_docs, embedding_model)
```

```
    retriever = vectordb.as_retriever(search_kwargs={"k": TOP_K})

    qa_chain = create_qa_chain(llm, retriever)

    os.remove(tmp_path)
    return "✅ PDF processed and indexed. Ready for questions."
```

Purpose:Takes raw PDF bytes, saves to a temporary file.Loads the PDF pages as documents.Splits documents into chunks for better semantic search.Creates a FAISS vector store from chunks using embeddings.Initializes a retriever and builds a QA chain.Cleans up the temporary file.

Input: **`pdf_file`** - raw PDF content as bytes.

Output: Status message indicating readiness.

iv). `Answer_question:`

```
def answer_question(question):
    global qa_chain
    if qa_chain is None:
        return "⚠️ Please upload and process a PDF first."
    try:
        result = qa_chain.invoke({"question": question})
        answer = truncate_context(result.get("answer", "⚠️ No answer found."))
        sources = result.get("sources", "No sources.")
        return f"🧠 **Answer:**\n{answer.strip()}"
    except Exception as e:
        return f"❌ Error: {str(e)}"
```

Purpose: Takes a user question and uses the initialized QA chain to generate an answer.Checks if a PDF has been processed first.Truncates the answer to max tokens.Returns answer and optionally sources (here, only answer shown).Catches and returns errors gracefully.

Input: **`question` (str)**

Output: Formatted answer string or error message.

MODULE - 3: Gradio UI for RAG PDF Chatbot (app.py) :

This code creates a web-based interface using **Gradio** for uploading PDFs, processing them into a searchable knowledge base, and then querying the content using a retrieval-augmented generation (RAG) pipeline built on **Flan-T5, FAISS, and LangChain.**

Key Components:

Imports:

```
import gradio as gr

from utils import process_pdf, answer_question
```

- **`gradio`:** For creating an interactive web UI.
- **`process_pdf`:** Function that takes a PDF file and processes it for retrieval.
- **`answer_question`:** Function that accepts a user query and returns an answer using the processed PDF context.

i.) UI Layout:

```
with gr.Blocks() as demo:

    gr.Markdown("## 🤖 RAG PDF Chatbot (Flan-T5 Small + FAISS + LangChain)")
```

- Creates a Gradio interface container **(`Blocks`)** and adds a markdown title.

ii.) PDF Upload Section:

```
with gr.Row():
    pdf_input = gr.File(label="📄 Upload PDF", type="binary")
    upload_btn = gr.Button("📥 Process PDF")
upload_output = gr.Textbox(label="📌 Status")
```

- **`pdf_input`**: File uploader widget accepting PDF files in binary mode.
- **`upload_btn`**: Button to trigger processing of the uploaded PDF.
- **`upload_output`**: Textbox to display status messages like "PDF processed" or error notifications.

iii.) Question Asking Section

```
question_input = gr.Textbox(label="💬 Ask a Question", placeholder="e.g., What are the key insights in section 2?")
ask_btn = gr.Button("🔍 Get Answer")
answer_output = gr.Textbox(label="🧠 Answer with Sources", lines=10)
```

- **`question_input`**: Text input box for user questions.
- **`ask_btn`**: Button to submit the question.
- **`answer_output`**: Textbox showing the answer from the QA chain.

iv.) Event Bindings:

```
upload_btn.click(process_pdf, inputs=pdf_input, outputs=upload_output)

ask_btn.click(answer_question, inputs=question_input, outputs=answer_output)
```

- Clicking the **Process PDF** button triggers **`process_pdf`** with the uploaded PDF file as input and displays the status in **`upload_output.`**
- Clicking the **Get Answer** button triggers **`answer_question`** with the question text as input and shows the answer in **`answer_output`**.

Launch

demo.launch(server_name="0.0.0.0", server_port=7860).Starts the Gradio web server, accessible on all network interfaces (for example, **`http://localhost:7860`**).Useful for local testing or deployment on a server.