🚀 Building a Simple RAG Chatbot Pipeline using Gemini API 🔑

📌 Introduction

This project demonstrates how to build a **basic RAG-based chatbot** using:

- Document/text data (PDFs and websites)
- Embedding and vector similarity
- Gemini (Google GenAI) for answering questions

The core idea of RAG is:

> "**Retrieve relevant information** from a data source, then use a **language model** to generate responses based on it."

✅ Step-by-Step Workflow Explained

1️⃣Install Dependencies

We install several Python libraries:

- `PyPDF2`, `pdf2image`, `pytesseract` – for PDF text and OCR extraction
- `beautifulsoup4`, `requests` – for scraping website text
- `faiss-cpu` – to store embeddings for fast similarity search
- `sentence-transformers` – to convert text chunks into embeddings
- `google-generativeai` – to access the Gemini API

```
pip install PyPDF2 beautifulsoup4 requests faiss-cpu
sentence-transformers google-generativeai
pip install pytesseract pdf2image
apt-get install poppler-utils tesseract-ocr
pip install pdfminer.six
```

2️⃣Upload PDF & Extract Text

The user is prompted to upload a PDF file. The text is extracted using Optical Character Recognition (OCR) from scanned images:

```
from pdf2image import convert_from_path
import pytesseract

def extract_text_from_scanned_pdf(pdf_path):
    ...
```

You can extract text from:

- **Digitally generated PDFs** (using PDFMiner or PyPDF2)
- **Scanned image PDFs** (using `pytesseract`)

3️⃣Scrape Website Content

A URL is scraped to fetch text (e.g., paragraphs) using BeautifulSoup.

```
def scrape_website(url):
    ...
```

This gives us additional context/data beyond the PDF.

4️⃣Preprocess and Chunk Text

The combined PDF and web text is split into chunks (~500 words each), because most LLMs (like Gemini) have input token limits.

```
def chunk_text(text, chunk_size=500):
    ...
```

This ensures better memory management and context segmentation.

5️⃣ Embed Chunks using Sentence Transformers

We use a pretrained model (MiniLM) from `sentence-transformers` to embed each chunk into a vector space.

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("all-MiniLM-L6-v2")
embeddings = model.encode(chunks)
```

This converts our textual chunks into numerical vectors for similarity comparison.

6️⃣ Store Embeddings in FAISS Index

FAISS is a library for efficient similarity search. We store the vector embeddings in FAISS so that we can later **search for the most relevant chunk** based on user query.

```
import faiss

index = faiss.IndexFlatL2(embeddings.shape[1])
index.add(np.array(embeddings))
```

7️⃣ Accept User Queries & Embed

When the user enters a query, we embed it similarly using the same SentenceTransformer model:

```
query_embedding = model.encode([user_query])
```

8️⃣Retrieve Relevant Chunks (Top-k Search)

We compare the query embedding with our FAISS index to get the top-k most similar document chunks:

```
D, I = index.search(query_embedding, k=3)
```

These chunks are then concatenated to form a **context** for the final generation.

9️⃣Generate Response using Gemini

We use the Google Generative AI (Gemini) model to generate the final answer based on retrieved chunks and the user's query.

```
from google import generativeai as genai

def generate_answer(prompt):
    ...
```

The prompt includes:

- Retrieved context (from chunks)
- User question

Example prompt:

```
Use the context below to answer the question:

[context from chunks]

Q: What is NLP?
A: ...
```