A Lightweight Multimodal AI Chatbot using Local LLM and Hybrid Data Sources

Start coding or generate with AI.

Step 1: Install Required Libraries

Since we're using Google Colab, install the necessary libraries:

pypdf – Extract text from PDFs

transformers – Use pre-trained language models

sentence-transformers – Convert text into embeddings

chromadb – Store embeddings in a vector database

langchain – Make retrieval and LLM integration easy

```
!pip install pypdf transformers sentence-transformers chromadb langchain
```

Show hidden output

Start coding or generate with AI.

Step 2: Upload and Read a Book/PDF

Colab allows file uploads using the files module.

Start coding or generate with AI.

```
!pip install PyPDF2
```

```
Collecting PyPDF2
  Downloading pypdf2-3.0.1-py3-none-any.whl.metadata (6.8 kB)
Downloading pypdf2-3.0.1-py3-none-any.whl (232 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 232.6/232.6 kB 14.7 MB/s eta 0:00:00
Installing collected packages: PyPDF2
Successfully installed PyPDF2-3.0.1
```

Start coding or generate with AI.

Extract text from the uploaded PDF:

Start coding or generate with AI.

```
import requests
from bs4 import BeautifulSoup
from google.colab import files
from PyPDF2 import PdfReader

# Function to extract text from uploaded PDFs (starting from page 3)
def extract_text_from_uploaded_pdfs():
    uploaded = files.upload()
    all_text = ""
    for filename in uploaded:
        try:
            reader = PdfReader(filename)
            for page in reader.pages[2:]:  # Skip first two pages
                text = page.extract_text()
                if text:
                    all_text += text + "\n"
        except Exception as e:
            print(f"❌ Could not read {filename}: {e}")
    return all_text

# Function to extract text from multiple URLs
def extract_text_from_urls():
    urls = input("Paste one or more URLs (comma separated): ").split(",")
    all_text = ""
    for url in urls:
        url = url.strip()
        try:
            response = requests.get(url)
            soup = BeautifulSoup(response.content, "html.parser")
            paragraphs = soup.find_all("p")
            for para in paragraphs:
                text = para.get_text().strip()
                if text:
                    all_text += text + "\n"
        except Exception as e:
            print(f"❌ Failed to extract from {url}: {e}")
    return all_text

# PDF text extraction using PyPDF2 directly from a specified file
def extract_text_from_pdf(pdf_path):
    text = ""
    try:
        with open(pdf_path, "rb") as file:
            reader = PdfReader(file)
```

```
            for page in reader.pages:
                page_text = page.extract_text()
                if page_text:
                    text += page_text + "\n"
    except Exception as e:
        print(f"❌ Error reading PDF: {e}")
    return text

# Prompt user to choose input method
print("📥 Choose your input source:")
print("1. Upload PDF(s)")
print("2. Enter URLs")
print("3. Both PDF(s) and URLs")
choice = input("Enter your choice (1/2/3): ").strip()

extracted_text = ""

if choice == "1":
    print("\n📄 Please upload your PDF file(s):")
    extracted_text = extract_text_from_uploaded_pdfs()

elif choice == "2":
    print("\n🌐 Please enter the URL(s):")
    extracted_text = extract_text_from_urls()

elif choice == "3":
    print("\n📄 Upload PDF file(s) first:")
    extracted_text += extract_text_from_uploaded_pdfs()
    print("\n🌐 Now enter the URL(s):")
    extracted_text += extract_text_from_urls()

else:
    print("❌ Invalid choice. Please enter 1, 2, or 3.")
```

```
📥 Choose your input source:
1. Upload PDF(s)
2. Enter URLs
3. Both PDF(s) and URLs
Enter your choice (1/2/3): 1

📄 Please upload your PDF file(s):
Choose Files hands-on-d…-science.pdf
• hands-on-data-science.pdf(application/pdf) - 16024943 bytes, last modified: 5/6/2025 - 100% done
Saving hands-on-data-science.pdf to hands-on-data-science.pdf
```

```
# Print preview of extracted text (optional)
print("\n✅ Preview of extracted text:")
print(extracted_text[:1000])  # Show first 1000 characters
```

```
✅ Preview of extracted text:
```

```
Hands-On Data Science and Python Machine
Learning
Copyright © 2017 Packt Publishing
All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or
transmitted in any form or by any means, without the prior written permission of thepublisher, except in the case of brief quotations embedded in critical arti
Every effort has been made in the preparation of this book to ensure the accuracy of the
information presented. However, the information contained in this book is sold withoutwarranty, either express or implied. Neither the author, nor Packt Publis
Packt Publishing has endeavored to provide trademark information about all of the
companies and products mentioned in this book by the appropriate use of capitals.However, Packt Publishing cannot guarantee the accuracy of this information.
First published:
```

Start coding or generate with AI.

Step 3: Preprocess Text (NLP Cleaning)

Before converting text into embeddings, clean it:

- Lowercasing
- Removing special characters
- Tokenization & Lemmatization

```
import re
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# Download necessary NLTK resources (only once)
nltk.download("punkt")
nltk.download("stopwords")
nltk.download("wordnet")
nltk.download("omw-1.4")
nltk.download("averaged_perceptron_tagger")
# Download the missing 'punkt_tab' resource
nltk.download('punkt_tab')

# Function to clean and preprocess extracted text
def clean_text(text):
    # Lowercase and remove special characters
    text = text.lower()
    text = re.sub(r"\W+", " ", text)

    # Tokenization
    tokens = word_tokenize(text)

    # Stopword removal
```

```
    stop_words = set(stopwords.words("english"))
    tokens = [word for word in tokens if word not in stop_words]

    # Lemmatization
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]

    return " ".join(tokens)

# Clean the previously extracted_text (from PDFs, URLs, or both)
cleaned_text = clean_text(extracted_text)

# Preview the cleaned result
print("\n🌸 Cleaned Text Preview:")
print(cleaned_text[:1000])  # Show first 1000 characters
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.

🌸 Cleaned Text Preview:
hand data science python machine learning copyright 2017 packt publishing right reserved part book may reproduced stored retrieval system transmitted form mear
```

Start coding or generate with AI.

Step 4: Convert Text into Embeddings & Store in a Vector DB

Use sentence-transformers to generate embeddings and store them in ChromaDB.

```
from sentence_transformers import SentenceTransformer
import chromadb

model = SentenceTransformer("all-MiniLM-L6-v2")  # Efficient embedding model

# Initialize ChromaDB
chroma_client = chromadb.PersistentClient(path="chroma_db")  # Stores embeddings persistently
collection = chroma_client.get_or_create_collection(name="documents")

# Split text into chunks
chunk_size = 500
```

```
chunks = [cleaned_text[i:i + chunk_size] for i in range(0, len(cleaned_text), chunk_size)]

# Store embeddings
for i, chunk in enumerate(chunks):
    embedding = model.encode(chunk).tolist()
    collection.add(ids=[str(i)], embeddings=[embedding], metadatas=[{"text": chunk}])

print("Text chunks stored in Vector DB!")
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
modules.json: 100%  349/349 [00:00<00:00, 23.0kB/s]
config_sentence_transformers.json: 100%  116/116 [00:00<00:00, 10.0kB/s]
README.md: 100%  10.5k/10.5k [00:00<00:00, 739kB/s]
sentence_bert_config.json: 100%  53.0/53.0 [00:00<00:00, 4.65kB/s]
config.json: 100%  612/612 [00:00<00:00, 52.5kB/s]
model.safetensors: 100%  90.9M/90.9M [00:01<00:00, 110MB/s]
tokenizer_config.json: 100%  350/350 [00:00<00:00, 11.8kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 6.91MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 26.1MB/s]
special_tokens_map.json: 100%  112/112 [00:00<00:00, 10.4kB/s]
config.json: 100%  190/190 [00:00<00:00, 13.4kB/s]
Text chunks stored in Vector DB!
```

Start coding or generate with AI.

Step 5: Retrieve Most Relevant Chunks

Now, when a user asks a question, we retrieve the most relevant chunks from ChromaDB.

```
def retrieve_relevant_chunks(query):
    query_embedding = model.encode(query).tolist()
    results = collection.query(query_embeddings=[query_embedding], n_results=3)  # Get top 3 matches
    return [res["text"] for res in results["metadatas"][0]]
```

```
query = "What is the content of this Site?"
retrieved_chunks = retrieve_relevant_chunks(query)
print("\n\n".join(retrieved_chunks))
```

```
sftvmu tus 6tfs hfout sftvmu dealing real world data 293 get following result see look legitimate scraper case actually malicious attack actually pretending l

ems little bit fishy let sdive little bit see actually looking blog page actuallygo file examine hand would see lot blog requestsdon actually user agent user a

website andhow much people spend example amazon concerned therelationship quickly page render much money people spend afterthat experience wanted know actual
```

Start coding or generate with AI.

Step 6: Pass Retrieved Chunks to LLM for Response

Model Name | Size | Accuracy | RAM Need | GPU Needed

1. GPT4All-J | 3–4B | Medium | 8GB | ❌
2. TinyLLaMA 1.1B | 1B | Low-Mid | 4–6GB | ❌
3. Mistral (Quantized) | 7B | Good | 8GB+ | ❌ (with quantization)
4. Phi-2 (Microsoft) | 2.7B | Great at reasoning | 8GB ❌

Install Required Libraries for Phi-2

```
!pip install transformers accelerate
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (4.52.3)
Requirement already satisfied: accelerate in /usr/local/lib/python3.11/dist-packages (1.7.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from transformers) (3.18.0)
Requirement already satisfied: huggingface-hub<1.0,>=0.30.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.32.2)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from transformers) (24.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.11/dist-packages (from transformers) (2024.11.6)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from transformers) (2.32.3)
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.21.1)
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.11/dist-packages (from transformers) (0.5.3)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.11/dist-packages (from transformers) (4.67.1)
Requirement already satisfied: psutil in /usr/local/lib/python3.11/dist-packages (from accelerate) (5.9.5)
Requirement already satisfied: torch>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from accelerate) (2.6.0+cu124)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.11/dist-packages (from huggingface-hub<1.0,>=0.30.0->transformers) (2025.3.2)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.11/dist-packages (from huggingface-hub<1.0,>=0.30.0->transformers) (4.13.2)
Requirement already satisfied: hf-xet<2.0.0,>=1.1.2 in /usr/local/lib/python3.11/dist-packages (from huggingface-hub<1.0,>=0.30.0->transformers) (1.1.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (3.1.6)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (12.4.127)
```

```
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (12.4.127)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (12.4.127)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (12.4.5.8)
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (11.2.1.3)
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (10.3.5.147)
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (11.6.1.9)
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (12.3.1.170)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (12.4.127)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (12.4.127)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages (from torch>=2.0.0->accelerate) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch>=2.0.0->accelerate) (1.3.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (3.4.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (2.4.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from requests->transformers) (2025.4.26)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch>=2.0.0->accelerate) (3.0.2)
```

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

# Load tokenizer and model (ensure you have a GPU runtime in Colab)
tokenizer = AutoTokenizer.from_pretrained("microsoft/phi-2")
model = AutoModelForCausalLM.from_pretrained("microsoft/phi-2", torch_dtype=torch.float16, device_map="auto")
```

```
tokenizer_config.json: 100%  7.34k/7.34k [00:00<00:00, 352kB/s]
vocab.json: 100%  798k/798k [00:00<00:00, 30.6MB/s]
merges.txt: 100%  456k/456k [00:00<00:00, 19.5MB/s]
tokenizer.json: 100%  2.11M/2.11M [00:00<00:00, 23.5MB/s]
added_tokens.json: 100%  1.08k/1.08k [00:00<00:00, 40.7kB/s]
special_tokens_map.json: 100%  99.0/99.0 [00:00<00:00, 5.89kB/s]
config.json: 100%  735/735 [00:00<00:00, 21.9kB/s]
model.safetensors.index.json: 100%  35.7k/35.7k [00:00<00:00, 1.80MB/s]
Fetching 2 files: 100%  2/2 [00:59<00:00, 59.63s/it]
model-00002-of-00002.safetensors: 100%  564M/564M [00:08<00:00, 24.1MB/s]
model-00001-of-00002.safetensors: 100%  5.00G/5.00G [00:59<00:00, 207MB/s]
Loading checkpoint shards: 100%  2/2 [00:00<00:00, 1.38it/s]
generation_config.json: 100%  124/124 [00:00<00:00, 4.54kB/s]
```

```
# Combine retrieved chunks into a context
context = "\n".join(retrieved_chunks)

# Prompt for LLM (RAG-style prompt)
prompt = f"""You are a helpful assistant. Based only on the context below, provide a concise and accurate answer to the question. Do not include anything else.:
Context:
{context}

Question:
{query}

Answer:"""


inputs = tokenizer(prompt, return_tensors="pt").to(model.device)
outputs = model.generate(**inputs, max_new_tokens=350, do_sample=True, temperature=0.7)
response = tokenizer.decode(outputs[0], skip_special_tokens=True)

# Print only the answer part
print(response.split("Answer:")[-1].strip())
```

```
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
The content of this site is a combination of web scraping data, legitimate browser activity, malicious attacks, user agent manipulation, search engine crawlers
```

Start coding or generate with AI.

1. max_new_tokens=350

👉 This means the model can generate up to 350 new words or tokens as a response.

2. do_sample=True

👉 This tells the model to add randomness while generating the answer.

3. temperature=0.7

👉 Controls how random or focused the output should be.

- 0.0 = very focused and deterministic (always gives same answer)
- 1.0 = very random (can be creative but sometimes silly)
- 0.7 = a good balance (slightly creative but still accurate)

So with temperature=0.7,

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

```
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
import os
import time
import textwrap  # Add once at top


# ✅ Load models once
embedding_model = SentenceTransformer("all-MiniLM-L6-v2")
tokenizer = AutoTokenizer.from_pretrained("microsoft/phi-2")
phi_model = AutoModelForCausalLM.from_pretrained("microsoft/phi-2", torch_dtype=torch.float16, device_map="auto")

# ✅ Feedback log
feedback_log = []

# ⛔ Simulated database (replace with actual retrieval DB)
# You need a vector DB like FAISS, Chroma, etc. Here it's assumed as `collection`
# Example dummy for placeholder (to avoid NameError during testing)
class DummyCollection:
    def query(self, query_embeddings, n_results):
        return {
            "metadatas": [[
                {"text": "Hyperparameter tuning is the process of choosing the best set of hyperparameters for a learning algorithm."},
                {"text": "It helps improve model performance by adjusting values such as learning rate, depth, or number of estimators."},
                {"text": "Common methods include Grid Search, Random Search, and Bayesian Optimization."}
            ]]
        }

collection = DummyCollection()

def clear_screen():
    os.system('cls' if os.name == 'nt' else 'clear')

# ✅ Chat loop
while True:
    clear_screen()
    print("💬 Welcome to Phi-2 Chatbot\n" + "=" * 40)
    query = input("🧑 You: ").strip()

    if query.lower() in ["byy", "stop", "exit", "quit"]:
        print("\n🤖 PhiBot: Thanks for chatting! 👋\n")
        break

    print("\n🤖 PhiBot is typing...", end="")
```

```
        time.sleep(1)

        # 🔍 Retrieve context
        query_embedding = embedding_model.encode(query).tolist()
        results = collection.query(query_embeddings=[query_embedding], n_results=3)
        retrieved_chunks = [res["text"] for res in results["metadatas"][0]]
        context = "\n".join(retrieved_chunks)

        # 🧠 Prompt
        prompt = f"""You are a helpful assistant. Based only on the context below, provide a concise and accurate answer to the question. Do not include anything else.

    Context:
    {context}

    Question:
    {query}

    Answer:"""

        # ✨ Generate answer
        inputs = tokenizer(prompt, return_tensors="pt").to(phi_model.device)
        outputs = phi_model.generate(
            **inputs,
            max_new_tokens=250,
            do_sample=True,
            temperature=0.7,
            top_p=0.9,
            repetition_penalty=1.1,
            eos_token_id=tokenizer.eos_token_id
        )

        response = tokenizer.decode(outputs[0], skip_special_tokens=True)
        if "Answer:" in response:
            answer = response.split("Answer:")[-1].strip()
        else:
            answer = response.strip()

        wrapped_answer = textwrap.fill(answer, width=80)
        print("\n🤖 Chatbot Answer:\n")
        print(wrapped_answer)
        print("-" * 60)


        # 💬 Feedback
        wrapped_prompt = textwrap.fill("👍 Was this answer helpful? (yes/no): ", width=80)
        feedback = input(wrapped_prompt).strip().lower()

        feedback_log.append({
           "question": query,
           "answer": answer,
           "feedback": feedback
```

```
})
```

```
Loading checkpoint shards: 100%                                    2/2 [00:01<00:00,  1.18s/it]
WARNING:accelerate.big_modeling:Some parameters are on the meta device because they were offloaded to the cpu.
💬 Welcome to Phi-2 Chatbot
========================================
🧑 You: what is machine learning

🤖 PhiBot is typing...Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

🤖 Chatbot Answer:

Machine Learning (ML) is an application of artificial intelligence that enables
computer systems to learn from data without being explicitly programmed. ML
algorithms use statistical techniques to find patterns in data and make
predictions or decisions based on those patterns. The goal of ML is to enable
computers to improve their performance on a specific task with experience.
Examples of ML applications include image recognition, natural language
processing, recommendation systems, and self-driving cars.
------------------------------------------------------------
👍 Was this answer helpful? (yes/no):no
💬 Welcome to Phi-2 Chatbot
========================================
🧑 You: deep learning

🤖 PhiBot is typing...Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

🤖 Chatbot Answer:

hyperparameter tuning
------------------------------------------------------------
👍 Was this answer helpful? (yes/no):no
💬 Welcome to Phi-2 Chatbot
========================================
🧑 You: byy

🤖 PhiBot: Thanks for chatting! 👋
```

Start coding or generate with AI.

```
==========================================================================================
```

```
!pip install -q sentence-transformers transformers chromadb streamlit pyngrok torch accelerate
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.3/44.3 kB 2.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.9/9.9 MB 18.1 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.9/6.9 MB 12.4 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 79.1/79.1 kB 5.4 MB/s eta 0:00:00
```

```
!pip install gradio
```

Show hidden output

```
!pip install gradio PyPDF2 beautifulsoup4 nltk sentence-transformers transformers chromadb
```

Show hidden output

```
import gradio as gr
import requests
from bs4 import BeautifulSoup
from PyPDF2 import PdfReader
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
import re, json
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer


# Download required NLTK resources
nltk.download("punkt")
nltk.download("stopwords")
nltk.download("wordnet")

# Load models and initialize ChromaDB
embedding_model = SentenceTransformer("all-MiniLM-L6-v2")
tokenizer = AutoTokenizer.from_pretrained("microsoft/phi-2")
phi_model = AutoModelForCausalLM.from_pretrained("microsoft/phi-2", torch_dtype=torch.float16, device_map="auto")

import chromadb
chroma_client = chromadb.PersistentClient(path="chroma_gradio_db")
collection = chroma_client.get_or_create_collection(name="documents")

# ----------- HELPER FUNCTIONS --------------
def extract_text_from_pdfs(pdf_files):
    all_text = ""
    for pdf in pdf_files:
        reader = PdfReader(pdf)
        for page in reader.pages[2:]:  # Skip first 2 pages
            text = page.extract_text()
            if text:
                all_text += text + "\n"
    return all_text

def extract_text_from_urls(url_input):
    urls = [url.strip() for url in url_input.split(",")]
    all_text = ""
```

```
    for url in urls:
        try:
            response = requests.get(url)
            soup = BeautifulSoup(response.content, "html.parser")
            paragraphs = soup.find_all("p")
            for para in paragraphs:
                text = para.get_text().strip()
                if text:
                    all_text += text + "\n"
        except Exception as e:
            all_text += f"\n❌ Failed to extract from {url}: {e}\n"
    return all_text

def clean_text(text):
    text = text.lower()
    text = re.sub(r"\W+", " ", text)
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words("english"))
    tokens = [word for word in tokens if word not in stop_words]
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    return " ".join(tokens)

def store_embeddings(cleaned_text):
    chunk_size = 500
    chunks = [cleaned_text[i:i+chunk_size] for i in range(0, len(cleaned_text), chunk_size)]
    for i, chunk in enumerate(chunks):
        embedding = embedding_model.encode(chunk).tolist()
        collection.add(ids=[str(i)], embeddings=[embedding], metadatas=[{"text": chunk}])
    return f"{len(chunks)} chunks stored in Vector DB!"

def answer_query(query):
    query_embedding = embedding_model.encode(query).tolist()
    results = collection.query(query_embeddings=[query_embedding], n_results=3)
    retrieved_chunks = [res["text"] for res in results["metadatas"][0]]
    context = "\n".join(retrieved_chunks)

    prompt = f"""You are a helpful assistant. Based only on the context below, provide a concise and accurate answer.

Context:
{context}

Question:
{query}

Answer:"""

    inputs = tokenizer(prompt, return_tensors="pt").to(phi_model.device)
    outputs = phi_model.generate(**inputs, max_new_tokens=350, do_sample=False)
    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
```

```
    answer = response.split("Answer:")[-1].strip() if "Answer:" in response else response.strip()
    return answer

# ----------- GRADIO WORKFLOW FUNCTIONS --------------

extracted_text_global = ""

def upload_and_extract(pdf_files, urls, show_preview):
    global extracted_text_global
    pdf_text = extract_text_from_pdfs(pdf_files) if pdf_files else ""
    url_text = extract_text_from_urls(urls) if urls else ""
    combined_text = pdf_text + url_text
    cleaned = clean_text(combined_text)
    extracted_text_global = cleaned
    store_result = store_embeddings(cleaned)
    preview = cleaned[:1000] if show_preview else "Preview not requested."
    return preview, store_result

def handle_query(user_query):
    answer = answer_query(user_query)
    return answer

# ----------- GRADIO UI --------------

with gr.Blocks(theme=gr.themes.Soft()) as demo:
    gr.Markdown(
        "<h1 style='text-align: center;'>🤖 AI-Powered QA Chatbot</h1>"
        "<p style='text-align: center;'>Upload PDFs or URLs and ask context-aware questions using ChromaDB & Phi-2 LLM</p>"
    )
# with gr.Blocks() as demo:
#     gr.Markdown("## 📚 AI Chatbot with Upload PDFs or URLs and ask context-aware questions using ChromaDB & Phi-2 LLM")

    with gr.Row():
        pdf_input = gr.File(file_types=[".pdf"], file_count="multiple", label="Upload PDFs")
        url_input = gr.Textbox(label="Paste URLs (comma-separated)")
        show_preview = gr.Checkbox(label="Show Cleaned Text Preview", value=True)

    extract_btn = gr.Button("Extract and Embed")
    output_preview = gr.Textbox(label="🧹 Cleaned Text Preview (optional)", lines=10)
    embed_status = gr.Textbox(label="📥 Embedding Status")

    with gr.Row():
        query_input = gr.Textbox(label="💬 Ask your question")
        query_btn = gr.Button("Get Answer")
        answer_output = gr.Textbox(label="🤖 Chatbot Answer", lines=5)

    extract_btn.click(upload_and_extract, inputs=[pdf_input, url_input, show_preview], outputs=[output_preview, embed_status])
    query_btn.click(handle_query, inputs=query_input, outputs=answer_output)

demo.launch()
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

Loading checkpoint shards: 100% 2/2 [00:00<00:00, 1.48it/s]

```
It looks like you are running Gradio on a hosted a Jupyter notebook. For the Gradio app to work, sharing must be enabled. Automatically setting `share=True` (y

Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://dc8228f58ed39b4db4.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working directory to deploy to Hug
```

Start coding or generate with AI.

```
def clear_embeddings():
    collection.delete(where={})  # Deletes all documents
    return "🗑️ All previous embeddings have been cleared!"
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

```
import gradio as gr
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import re
from PyPDF2 import PdfReader
import requests
from bs4 import BeautifulSoup
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
import chromadb

# Download required NLTK resources
nltk.download("punkt")
nltk.download("stopwords")
nltk.download("wordnet")

# Load models and initialize ChromaDB
embedding_model = SentenceTransformer("all-MiniLM-L6-v2")
tokenizer = AutoTokenizer.from_pretrained("microsoft/phi-2")
phi_model = AutoModelForCausalLM.from_pretrained(
    "microsoft/phi-2", torch_dtype=torch.float16, device_map="auto"
)

chroma_client = chromadb.PersistentClient(path="chroma_gradio_db")
collection = chroma_client.get_or_create_collection(name="documents")

# ----------- HELPER FUNCTIONS --------------
def extract_text_from_pdfs(pdf_files):
    all_text = ""
```

```
    for pdf in pdf_files:
        reader = PdfReader(pdf)
        for page in reader.pages[2:]:  # Skip first 2 pages
            text = page.extract_text()
            if text:
                all_text += text + "\n"
    return all_text

def extract_text_from_urls(url_input):
    urls = [url.strip() for url in url_input.split(",")]
    all_text = ""
    for url in urls:
        try:
            response = requests.get(url)
            soup = BeautifulSoup(response.content, "html.parser")
            paragraphs = soup.find_all("p")
            for para in paragraphs:
                text = para.get_text().strip()
                if text:
                    all_text += text + "\n"
        except Exception as e:
            all_text += f"\n❌ Failed to extract from {url}: {e}\n"
    return all_text

def clean_text(text):
    text = text.lower()
    text = re.sub(r"\W+", " ", text)
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words("english"))
    tokens = [word for word in tokens if word not in stop_words]
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    return " ".join(tokens)

def store_embeddings(cleaned_text):
    chunk_size = 500
    chunks = [cleaned_text[i:i+chunk_size] for i in range(0, len(cleaned_text), chunk_size)]
    for i, chunk in enumerate(chunks):
        embedding = embedding_model.encode(chunk).tolist()
        collection.add(ids=[str(i)], embeddings=[embedding], metadatas=[{"text": chunk}])
    return f"✅ {len(chunks)} chunks stored in Vector DB!"

def answer_query(query):
    query_embedding = embedding_model.encode(query).tolist()
    results = collection.query(query_embeddings=[query_embedding], n_results=3)
    retrieved_chunks = [res["text"] for res in results["metadatas"][0]]
    context = "\n".join(retrieved_chunks)

    prompt = f"""You are a helpful assistant. Based only on the context below, provide a concise and accurate answer.

Context:
```

```
    {context}

    Question:
    {query}

    Answer:"""

        inputs = tokenizer(prompt, return_tensors="pt").to(phi_model.device)
        outputs = phi_model.generate(**inputs, max_new_tokens=350, do_sample=False)
        response = tokenizer.decode(outputs[0], skip_special_tokens=True)

        answer = response.split("Answer:")[-1].strip() if "Answer:" in response else response.strip()
        return answer

# ----------- GRADIO WORKFLOW FUNCTIONS --------------
extracted_text_global = ""

def upload_and_extract(pdf_files, urls, show_preview):
    global extracted_text_global
    pdf_text = extract_text_from_pdfs(pdf_files) if pdf_files else ""
    url_text = extract_text_from_urls(urls) if urls else ""
    combined_text = pdf_text + url_text
    cleaned = clean_text(combined_text)
    extracted_text_global = cleaned
    store_result = store_embeddings(cleaned)
    preview = cleaned[:1000] if show_preview else "✅ Preview skipped."
    return preview, store_result

def handle_query(user_query):
    return answer_query(user_query)

# ----------- ENHANCED GRADIO UI --------------
with gr.Blocks(theme=gr.themes.Soft()) as demo:
    gr.Markdown("# 🤖 AI Chatbot: PDF / URL RAG-powered QA System with (ChromaDB + Phi-2 LLM).")

    with gr.Accordion("📥 Upload Data (PDFs / URLs)", open=True):
        with gr.Row():
            pdf_input = gr.File(file_types=[".pdf"], file_count="multiple", label="Upload PDFs")
            url_input = gr.Textbox(label="Paste URLs (comma-separated)")
        show_preview = gr.Checkbox(label="Show Cleaned Text Preview", value=True)
        extract_btn = gr.Button("🚀 Extract and Embed Text")
        output_preview = gr.Textbox(label="🧹 Cleaned Text Preview", lines=10, interactive=False)
        embed_status = gr.Textbox(label="📥 Embedding Status", interactive=False)

    with gr.Accordion("💬 Ask Questions", open=True):
        query_input = gr.Textbox(label="Type your question here")
        query_btn = gr.Button("🎯 Get Answer")
        answer_output = gr.Textbox(label="🤖 AI Answer", lines=5, interactive=False)

    extract_btn.click(
        upload_and_extract,
```

```
            inputs=[pdf_input, url_input, show_preview],
            outputs=[output_preview, embed_status]
        )

        query_btn.click(
            handle_query,
            inputs=query_input,
            outputs=answer_output
        )

    demo.launch()
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
Loading checkpoint shards: 100%                                    2/2 [00:19<00:00,  7.98s/it]
It looks like you are running Gradio on a hosted a Jupyter notebook. For the Gradio app to work, sharing must be enabled. Automatically setting `share=True` (y

Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://0ded54d3fad2a3d1e2.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working directory to deploy to Hug
```

gradio

No interface is running right now

```
import shutil
import os

chroma_path = "/content/chroma_gradio_db"  # or "chroma_gradio_db" depending on your setup

if os.path.exists(chroma_path):
    shutil.rmtree(chroma_path)
    print(f"✅ '{chroma_path}' deleted successfully.")
```

```
else:
    print(f"⚠️ '{chroma_path}' does not exist.")
```

```
✅ '/content/chroma_gradio_db' deleted successfully.
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

```
import gradio as gr
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import re
from PyPDF2 import PdfReader
import requests
from bs4 import BeautifulSoup
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
import chromadb
import os
import traceback

# Download required NLTK resources
nltk.download("punkt")
nltk.download("stopwords")
nltk.download("wordnet")

# Ensure ChromaDB folder exists
if not os.path.exists("chroma_gradio_db"):
    os.makedirs("chroma_gradio_db")

# Load models
# Load embedding model onto CPU
embedding_model = SentenceTransformer("all-MiniLM-L6-v2", device='cpu')
tokenizer = AutoTokenizer.from_pretrained("microsoft/phi-2")
phi_model = AutoModelForCausalLM.from_pretrained(
    "microsoft/phi-2",
    torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32,
    device_map="auto"
)

# Initialize ChromaDB
```

```
chroma_client = chromadb.PersistentClient(path="chroma_gradio_db")
collection = chroma_client.get_or_create_collection(name="documents")

# ----------- Helper Functions --------------
def extract_text_from_pdfs(pdf_files):
    all_text = ""
    for pdf in pdf_files:
        try:
            reader = PdfReader(pdf.name)  # Use .name to read file path
            for page in reader.pages[2:]:  # Skip first 2 pages
                text = page.extract_text()
                if text:
                    all_text += text + "\n"
        except Exception as e:
            all_text += f"\n❌ Failed to extract from {pdf.name}: {e}\n"
    return all_text

def extract_text_from_urls(url_input):
    if not url_input:
        return ""
    urls = [url.strip() for url in url_input.split(",") if url.strip()]
    all_text = ""
    for url in urls:
        try:
            response = requests.get(url)
            soup = BeautifulSoup(response.content, "html.parser")
            paragraphs = soup.find_all("p")
            for para in paragraphs:
                text = para.get_text().strip()
                if text:
                    all_text += text + "\n"
        except Exception as e:
            all_text += f"\n❌ Failed to extract from {url}: {e}\n"
    return all_text

def clean_text(text):
    text = text.lower()
    text = re.sub(r"\W+", " ", text)
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words("english"))
    tokens = [word for word in tokens if word not in stop_words]
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    return " ".join(tokens)

def store_embeddings(cleaned_text):
    if not cleaned_text.strip():
        return "❌ No valid text found to embed."

    collection.delete(where={})  # Clear previous documents
```

```
    chunk_size = 500
    chunks = [cleaned_text[i:i+chunk_size] for i in range(0, len(cleaned_text), chunk_size)]
    for i, chunk in enumerate(chunks):
        embedding = embedding_model.encode(chunk).tolist()
        collection.add(ids=[str(i)], embeddings=[embedding], metadatas=[{"text": chunk}])
    return f"✅ {len(chunks)} chunks stored in Vector DB!"

def answer_query(query):
    query_embedding = embedding_model.encode(query).tolist()
    results = collection.query(query_embeddings=[query_embedding], n_results=3)
    retrieved_chunks = [res["text"] for res in results["metadatas"][0]]
    context = "\n".join(retrieved_chunks)

    prompt = f"""You are a helpful assistant. Based only on the context below, provide a concise and accurate answer.

Context:
{context}

Question:
{query}

Answer:"""

    inputs = tokenizer(prompt, return_tensors="pt").to(phi_model.device)
    outputs = phi_model.generate(**inputs, max_new_tokens=150, do_sample=False)
    response = tokenizer.decode(outputs[0], skip_special_tokens=True)
    answer = response.split("Answer:")[-1].strip() if "Answer:" in response else response.strip()
    return answer

# ----------- Gradio Workflow --------------
extracted_text_global = ""

def upload_and_extract(pdf_files, urls, show_preview):
    global extracted_text_global
    try:
        pdf_text = extract_text_from_pdfs(pdf_files) if pdf_files else ""
        url_text = extract_text_from_urls(urls) if urls else ""
        combined_text = pdf_text + url_text
        cleaned = clean_text(combined_text)

        if not cleaned.strip():
            return "❌ No valid content extracted.", "❌ Nothing to embed."

        extracted_text_global = cleaned
        store_result = store_embeddings(cleaned)
        preview = cleaned[:1000] if show_preview else "✅ Preview skipped."
        return preview, store_result
    except Exception as e:
        return "❌ Error occurred during extraction", traceback.format_exc()

def handle_query(user_query):
```

```
    try:
        # Check if any documents are in the collection before querying
        if collection.count() == 0:
            return "Please upload and embed documents first."
        return answer_query(user_query)
    except Exception as e:
        return f"❌ Error: {str(e)}"

# ----------- Enhanced Gradio UI --------------
with gr.Blocks(theme=gr.themes.Soft()) as demo:
    gr.Markdown(
        "<h1 style='text-align: center;'>🤖 AI-Powered QA Chatbot</h1>"
        "<p style='text-align: center;'>Upload PDFs or URLs and ask context-aware questions using ChromaDB & Phi-2 LLM</p>"
    )

    with gr.Tab("📁 Upload & Embed"):
        with gr.Row():
            pdf_input = gr.File(file_types=[".pdf"], file_count="multiple", label="Upload PDFs 📄")
            url_input = gr.Textbox(label="Enter URLs (comma-separated) 🌐", placeholder="https://example.com, https://abc.com")
        show_preview = gr.Checkbox(label="🔍 Show Cleaned Text Preview", value=True)
        extract_btn = gr.Button("🚀 Extract & Embed Text")
        with gr.Column():
            output_preview = gr.Textbox(label="🧹 Cleaned Text Preview", lines=10, interactive=False)
            embed_status = gr.Textbox(label="📦 Embedding Status", interactive=False)

    with gr.Tab("💬 Ask a Question"):
        query_input = gr.Textbox(label="❓ Ask your question", placeholder="e.g., What is the main topic of the documents?")
        query_btn = gr.Button("🎯 Get Answer")
        answer_output = gr.Textbox(label="🤖 Answer from AI", lines=5, interactive=False)

    extract_btn.click(
        upload_and_extract,
        inputs=[pdf_input, url_input, show_preview],
        outputs=[output_preview, embed_status]
    )

    query_btn.click(
        handle_query,
        inputs=query_input,
        outputs=answer_output
    )

demo.launch()
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
Loading checkpoint shards: 100%                                    2/2 [00:29<00:00, 29.85s/it]
WARNING:accelerate.big_modeling:Some parameters are on the meta device because they were offloaded to the disk and cpu.
It looks like you are running Gradio on a hosted a Jupyter notebook. For the Gradio app to work, sharing must be enabled. Automatically setting `share=True` (y

Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://3a61d44b6e3818a6f9.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the working directory to deploy to Hug
```

gradio