```
!pip install gradio transformers sentence-transformers faiss-cpu PyMuPDF accelerate

import gradio as gr
import fitz  # PyMuPDF
import torch
import numpy as np
from typing import List, Tuple
from transformers import AutoTokenizer, AutoModelForCausalLM
from sentence_transformers import SentenceTransformer
import faiss, re, logging
from pathlib import Path
from dataclasses import dataclass

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class DocumentChunk:
    text: str
    source_file: str
    page_number: int
    chunk_id: int

class StudyMateSystem:
    def __init__(self):
        self.model = None
        self.tokenizer = None
        self.embedding_model = None
```

```
        self.vector_store = None
        self.document_chunks: List[DocumentChunk] = []
        self.chunk_embeddings = None
        self.is_initialized = False

    def initialize_models(self, progress_callback=None, use_small_model=True):
        try:
            if progress_callback:
                progress_callback("🔄 Loading language model... (this may take 1-3 mins)")
            device = "cuda" if torch.cuda.is_available() else "cpu"

            model_name = "google/gemma-2b-it" if use_small_model else "ibm-granite/granite-3.3-2b-instruct"
            self.tokenizer = AutoTokenizer.from_pretrained(model_name)
            self.model = AutoModelForCausalLM.from_pretrained(
                model_name,
                torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32
            ).to(device)

            if progress_callback:
                progress_callback("🔄 Loading embedding model...")
            self.embedding_model = SentenceTransformer('all-MiniLM-L6-v2')

            self.is_initialized = True
            if progress_callback:
                progress_callback("✅ Models loaded successfully!")
            logger.info("All models initialized successfully.")
        except Exception as e:
```

```
            error_msg = f"Error initializing models: {str(e)}"
            logger.error(error_msg)
            if progress_callback:
                progress_callback(error_msg)
            raise

    def extract_text_from_pdf(self, pdf_path: str) -> List[Tuple[str, int]]:
        try:
            doc = fitz.open(pdf_path)
            pages_text = []
            for page_num in range(len(doc)):
                text = self._clean_text(doc[page_num].get_text())
                if text.strip():
                    pages_text.append((text, page_num + 1))
            doc.close()
            return pages_text
        except Exception as e:
            logger.error(f"Error extracting text from {pdf_path}: {str(e)}")
            return []

    def _clean_text(self, text: str) -> str:
        text = re.sub(r'\s+', ' ', text)
        text = re.sub(r'[^\w\s.,!?;:()\-\'"]', ' ', text)
        lines = text.split('\n')
        cleaned_lines = [line.strip() for line in lines if len(line.strip()) > 10]
        return ' '.join(cleaned_lines).strip()

    def chunk_text(self, text: str, chunk_size: int = 500, overlap: int = 50) -> List[str]:
```

```
        words = text.split()
        chunks = []
        for i in range(0, len(words), chunk_size - overlap):
            chunk_text = ' '.join(words[i:i + chunk_size])
            if chunk_text.strip():
                chunks.append(chunk_text)
        return chunks

    def process_pdfs(self, pdf_files: List[str], progress_callback=None) -> str:
        try:
            if not self.is_initialized:
                return "❌ Models not initialized."
            if not pdf_files:
                return "❌ No PDF files provided."

            self.document_chunks = []
            all_texts = []

            for idx, pdf_file in enumerate(pdf_files):
                if progress_callback:
                    progress_callback(f"📄 Processing PDF {idx+1}/{len(pdf_files)}: {Path(pdf_file).name}")
                pages_text = self.extract_text_from_pdf(pdf_file)
                for page_text, page_num in pages_text:
                    for chunk_text in self.chunk_text(page_text):
                        chunk = DocumentChunk(chunk_text, Path(pdf_file).name, page_num, len(self.document_chunks))
                        self.document_chunks.append(chunk)
```

```
                all_texts.append(chunk_text)

        if not all_texts:
            return "❌ No text extracted."

        if progress_callback:
            progress_callback("🔄 Creating embeddings...")
        self.chunk_embeddings = self.embedding_model.encode(all_texts)
        dimension = self.chunk_embeddings.shape[1]
        self.vector_store = faiss.IndexFlatIP(dimension)
        faiss.normalize_L2(self.chunk_embeddings)
        self.vector_store.add(self.chunk_embeddings.astype('float32'))

        return f"✅ Processed {len(pdf_files)} PDFs\n📄 Extracted {len(self.document_chunks)} chunks"
    except Exception as e:
        return f"❌ Error: {str(e)}"

def retrieve_relevant_chunks(self, query: str, top_k=5):
    if not self.vector_store:
        return []
    query_embedding = self.embedding_model.encode([query])
    faiss.normalize_L2(query_embedding)
    scores, indices = self.vector_store.search(query_embedding.astype('float32'), top_k)
    return [self.document_chunks[i] for i, s in zip(indices[0], scores[0]) if s > 0.3]

def generate_answer(self, query: str, context_chunks: List[DocumentChunk]) -> str:
```

```
        if not context_chunks:
            return "No relevant information found."
        context_text = "\n\n".join([
            f"[{chunk.source_file} | Page {chunk.page_number}]\n{chunk.text}"
            for chunk in context_chunks[:3]
        ])
        prompt = f"Context:\n{context_text}\n\nQuestion: {query}\nAnswer:"
        inputs = self.tokenizer(prompt, return_tensors="pt").to(self.model.device)
        with torch.no_grad():
            outputs = self.model.generate(**inputs, max_new_tokens=300)
        return self.tokenizer.decode(outputs[0], skip_special_tokens=True)

    def answer_question(self, query: str):
        relevant_chunks = self.retrieve_relevant_chunks(query)
        answer = self.generate_answer(query, relevant_chunks)
        sources = "\n".join([f"- {chunk.source_file} (Page {chunk.page_number})" for chunk
in relevant_chunks[:3]])
        return answer, sources

study_mate = StudyMateSystem()

def initialize_system():
    study_mate.initialize_models(lambda msg: init_status.update(value=msg))
    return "✅ System Ready!"

def process_uploaded_files(files):
    file_paths = [f.name for f in files]
```

```
    return study_mate.process_pdfs(file_paths, lambda msg: upload_status.update(value=msg))

def ask_question(question):
    answer, sources = study_mate.answer_question(question)
    return answer, sources

with gr.Blocks() as interface:
    gr.Markdown("# 🎓 StudyMate - AI PDF Q&A")
    init_status = gr.Textbox(label="Initialization Status", value="Click button to load models", interactive=False)
    init_btn = gr.Button("Initialize System")
    init_btn.click(initialize_system, outputs=[init_status])

    file_upload = gr.File(label="Upload PDFs", file_types=[".pdf"], file_count="multiple")
    upload_status = gr.Textbox(label="Upload Status", interactive=False, lines=3)
    upload_btn = gr.Button("Process PDFs")
    upload_btn.click(process_uploaded_files, inputs=[file_upload], outputs=[upload_status])

    question = gr.Textbox(label="Ask a Question")
    ask_btn = gr.Button("Get Answer")
    answer_output = gr.Textbox(label="Answer", lines=8)
    sources_output = gr.Textbox(label="Sources", lines=5)
    ask_btn.click(ask_question, inputs=[question], outputs=[answer_output, sources_output])

interface.launch(share=True, inline=True)
```