# Nestlé HR Assistant Final Code Implementation

import torch  
from transformers import LlamaForCausalLM, LlamaTokenizer, StoppingCriteria, StoppingCriteriaList  
import gradio as gr  
from PyPDF2 import PdfReader  
import numpy as np  
from huggingface\_hub import login  
  
# Step 1: Login to Hugging Face (ensure you have the appropriate access)  
login() # This will prompt you to enter your Hugging Face token  
  
# Load LLaMA model and tokenizer  
model\_name = "meta-llama/Meta-Llama-3.1-8B"  
tokenizer = LlamaTokenizer.from\_pretrained(model\_name)  
tokenizer.pad\_token = tokenizer.eos\_token # Set pad\_token to eos\_token  
model = LlamaForCausalLM.from\_pretrained(model\_name, torch\_dtype=torch.float16).to("cuda")  
  
# Step 2: Load and Process HR Policy Documents  
def load\_and\_split\_pdf(pdf\_path):  
 Loads and splits the PDF document into manageable chunks of text.  
 reader = PdfReader(pdf\_path)  
 text\_chunks = []  
 for page in reader.pages:  
 text = page.extract\_text()  
 if text:  
 text\_chunks.extend(text.split("  
  
")) # Split into chunks by paragraphs  
 return text\_chunks  
  
# Step 3: Generate Embeddings using LLaMA  
def generate\_embeddings(text):  
 Generate embeddings using LLaMA model.  
 inputs = tokenizer(text, return\_tensors="pt", truncation=True, padding=True).to("cuda")  
 with torch.no\_grad():  
 outputs = model(\*\*inputs, output\_hidden\_states=True)  
 hidden\_states = outputs.hidden\_states[-1] # Get the last hidden state  
 return hidden\_states.mean(dim=1).cpu().numpy() # Return the mean of hidden states  
  
# Step 4: Cosine Similarity Function  
def cosine\_similarity(a, b):  
 Calculate the cosine similarity between two vectors.  
 a = a.flatten() # Flatten the vector to ensure it's one-dimensional  
 b = b.flatten() # Flatten the vector to ensure it's one-dimensional  
 return np.dot(a, b) / (np.linalg.norm(a) \* np.linalg.norm(b))  
  
# Custom stopping criteria to stop generation at the end of a complete thought  
class IntelligentStoppingCriteria(StoppingCriteria):  
 def \_\_init\_\_(self, tokenizer, initial\_length, threshold=0.9):  
 self.tokenizer = tokenizer  
 self.initial\_length = initial\_length  
 self.threshold = threshold # Confidence threshold for stopping  
  
 def \_\_call\_\_(self, input\_ids, scores, \*\*kwargs):  
 # Decode the generated text  
 generated\_text = self.tokenizer.decode(input\_ids[0], skip\_special\_tokens=True)  
   
 # Stop if we reached the end of a sentence and the score is above the threshold  
 if len(generated\_text) > self.initial\_length and generated\_text.endswith('.'):  
 if scores is not None:  
 avg\_score = torch.mean(torch.stack(scores)).item() # Calculate average confidence  
 if avg\_score > self.threshold:  
 return True  
 return False  
  
# Step 5: Build Question-Answering System  
def get\_best\_answer(user\_query, text\_chunks, embeddings):  
 Finds the most relevant text chunk based on user query and generates a response.  
 query\_embedding = generate\_embeddings(user\_query)  
 similarities = [cosine\_similarity(query\_embedding, emb) for emb in embeddings]  
 best\_chunk\_index = np.argmax(similarities)  
 best\_chunk = text\_chunks[best\_chunk\_index]  
   
 # Refine prompt to guide the model  
 prompt = f"Answer the following query concisely but completely based on the provided context:  
  
{user\_query}  
  
Context: {best\_chunk}"  
  
 # Generate response using the best text chunk  
 inputs = tokenizer(prompt, return\_tensors="pt", truncation=False, padding=True).to("cuda")  
 stopping\_criteria = StoppingCriteriaList([IntelligentStoppingCriteria(tokenizer, len(prompt))])  
  
 with torch.no\_grad():  
 response = model.generate(  
 input\_ids=inputs.input\_ids,  
 attention\_mask=inputs.attention\_mask,  
 max\_new\_tokens=500, # Allow sufficient tokens but let stopping criteria control the length  
 stopping\_criteria=stopping\_criteria,  
 pad\_token\_id=tokenizer.eos\_token\_id  
 )  
   
 generated\_text = tokenizer.decode(response[0], skip\_special\_tokens=True)  
 return generated\_text.strip()  
  
# Step 6: Design Gradio Interface  
def chatbot\_interface(user\_query):  
 Handles the interaction between the user and the chatbot.  
 try:  
 text\_chunks = load\_and\_split\_pdf("nestle\_hr\_policy.pdf")  
 embeddings = [generate\_embeddings(chunk) for chunk in text\_chunks]  
 answer = get\_best\_answer(user\_query, text\_chunks, embeddings)  
 return answer  
 except Exception as e:  
 print(f"An error occurred: {e}")  
 return "Error: Something went wrong. Please try again later."  
  
# Step 7: Gradio Interface for User Interaction  
interface = gr.Interface(  
 fn=chatbot\_interface,   
 inputs=gr.Textbox(lines=2, placeholder="Enter your HR-related question here..."),   
 outputs="text",   
 title="Nestlé HR Assistant",  
 description="Ask any question related to Nestlé's HR policies, and the assistant will provide you with accurate information.",  
 live=False  
)  
  
# Launch the interface  
interface.launch()